AFRL-RI-RS-TR-2009-144
Final Technical Report
May 2009

HIGH ASSURANCE VIRTUALIZATION ENGINE (HAVEN)

Polytechnic Institute of NYU

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-144 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
LOK YAN
Work Unit Manager

/s/
EDWARD J. JONES, DeputyChief
Advanced Computing Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* MAY 09	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jan 08 – Dec 08

4. TITLE AND SUBTITLE HIGH ASSURANCE VIRTUALIZATION ENGINE (HAVEN)	**5a. CONTRACT NUMBER** N/A
	5b. GRANT NUMBER FA8750-08-1-0068
	5c. PROGRAM ELEMENT NUMBER 62702F
6. AUTHOR(S) Ramesh Karri, Nasir Memon, Vikram Padman, and Pratik Mathur	**5d. PROJECT NUMBER** 459T
	5e. TASK NUMBER HA
	5f. WORK UNIT NUMBER VE
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Polytechnic Institute of NYU 6 Metrotech Center Brooklyn, NY 11201	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RITB 525 Brooks Rd. Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** N/A
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2009-144

12. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# 88ABW-2009-2183

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report describes the research results of the High Assurance Virtualization ENgine. HAVEN is an FPGA-based virtualization technology that implements much of the traditional hypervisor functionality in FPGAs instead of in software. There are two main results: A Secure Virtual I/O Manager (SIM) and a Secure Memory Manager (SMM). The Secure Virtual I/O Manager implements a virtual PCI controller along with multiple virtual Network Interface Cards (NIC) in conjunction with independent data buffers on a single FPGA. The CPU sees multiple NIC even though there is only one true physical card. The SMM registers a memory range with the CPU and ensures that all memory managed by the SMM is encrypted and only decrypted when it is moved to the CPU cache.

15. SUBJECT TERMS

Virtualization, FPGA, High-Assurance, Hypervisor, Virtual PCI, Virtual Network Interface Card (NIC), Memory, Encryption

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Lok K. Yan
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	52	**19b. TELEPHONE NUMBER (*Include area code)*** N/A

TABLE OF CONTENTS

LIST OF FIGURES

1 SUMMARY

Virtualization technology has been around since the late 1960's. Initially, it was conceived to maximize utilization of expensive hardware by running multiple instances of an operating system (OS) using virtual machines (VM). In the last decade, virtualization has become popular due to its cost and space saving advantages.

For efficient virtualization, two key features must be supported by the underlying hardware. Firstly, it should support isolation of VMs from one another. Secondly, it should support a virtual input/output (I/O) system. Most mainframe computer architectures traditionally provide hardware support for virtualization. Conversely, most personal computers are based on the x86 architecture, which does not provide hardware support for virtualization. Recently, major x86 Central Processing Unit (CPU) manufacturers have added limited hardware support for isolation while relying on software emulated virtual I/O subsystems. Thus, current virtualization architectures are not able to provide the high level of assurance and performance needed for mission critical applications. Moreover, several performance and security issues have been identified in current software-based virtualization architectures for x86 architectures.

These security and performance issues arise from the fact that the current virtualization architecture depends on a central VM0 to provide critical I/O services. The High Assurance Virtualization ENgine (HAVEN) architecture removes critical functionality from the central VM0 and moves it into a secure Field Programmable Gate Array (FPGA) based virtualization engine. This forms virtualized hardware, which provides a key security design principle commonly referred to as compartmentalization.

The goal of the HAVEN was to create a FPGA-based virtualization engine that addresses the reliability, performance and security limitations of current software-based virtualization technologies (Xen, in the current implementation). When inserted into commodity desktop computing platforms, HAVEN assumes many of today's hypervisor-functions, adding security and functionality.

At the core of HAVEN architecture is a new virtual I/O subsystem for I/O device isolation and memory subsystem to protect a virtual machine's memory from the host operating system. Both subsystems are implemented on FPGA development platforms.

I/O virtualization addresses the specific issue of how software-based virtualization solutions share the I/O resources. In Xen, each guest operating system is assigned a software emulated virtual I/O device that is separate from all other guests. However, on the host side, the memory space allocated to emulated I/O devices and the data buffers on the physical I/O device are actually shared among all guest virtual machines. HAVEN implements the hardware assisted virtual I/O device such that each guest machine is allocated its own memory and buffers, minimizing the possibility of cross-contamination of data between guest VMs. HAVEN's virtual I/O manager exports multiple Peripheral Component Interconnect (PCI) configuration registers to the system to create virtual I/O devices.

The memory isolation block contains mechanisms to fully encrypt (using separate keys) the memory spaces of individual guest operating system instances with the associated key-management infrastructure that supports hibernation of virtual machines.

2 INTRODUCTION

Virtualization consolidates underutilized servers and workstations while maintaining isolation. For software developers, virtualization provides an environment to develop, test, and debug system software such as kernel and device drivers. Traditionally, separate computers were required to develop and test system software. Virtualization also allows developers to test the reliability of an application by simulating hardware bottlenecks and failures. In theory [1], it is possible to create isolated VMs to test untrustworthy applications without exposing trusted applications along with critical data [2]. Finally, VMs may also be used to trap malicious users and observe their activities [3].

2.1 Virtualization Architectures and Limitations

There are many definitions for a VM. Popek [1] defines a VM as an efficient, isolated duplicate of the real machine. Goldberg [4] defines it as a hardware-software duplicate of a real computer system in which a statistically dominant subset of the virtual processor’s instructions execute on the host processor in native mode. Most computers today are based on the x86 architecture developed by IBM. Unlike mainframe computers that were designed for large scale commercial purposes, x86-based computers were primarily designed to be cost effective and hence did not support virtualization. Although several software-based virtualization techniques were subsequently developed, they entail significant performance penalties. Irvine discusses the poor performance of software-based virtualization in his analysis of the Intel Pentium’s ability to support a secure virtual machine monitor [2]. Recently, x86 architectures have evolved to support virtualization.

Figure 1 is a high level diagram of the interactions in the current virtualization architecture.

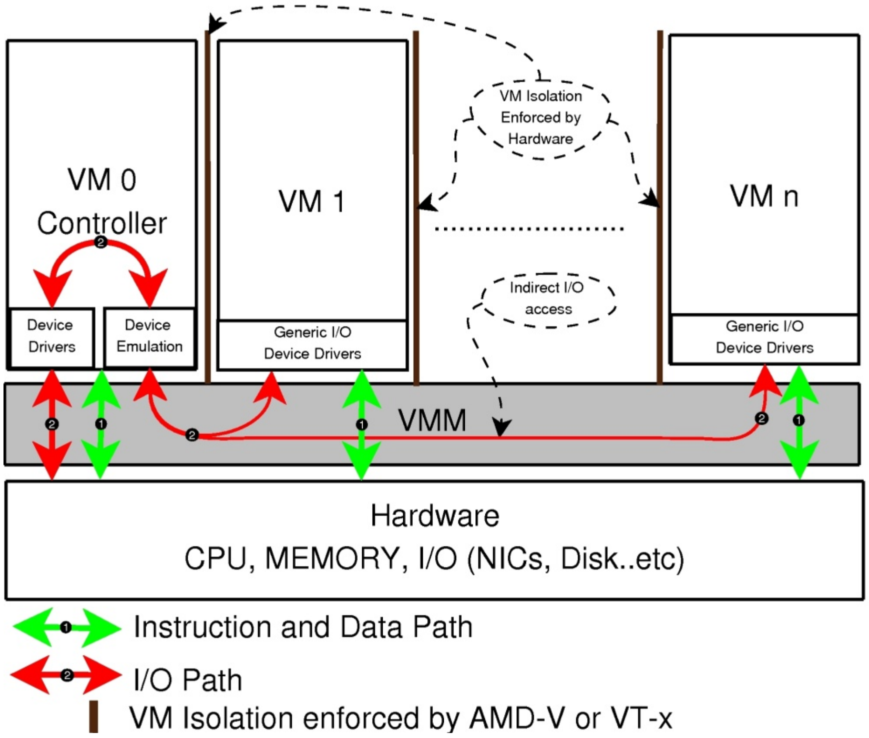

VM 0
Controller
Device Drivers
Device Emulation
VM 1
Generic I/O Device Drivers
VM Isolation Enforced by Hardware
Indirect I/O access
VM n
Generic I/O Device Drivers
VMM
Hardware
CPU, MEMORY, I/O (NICs, Disk..etc)
Instruction and Data Path
I/O Path
VM Isolation enforced by AMD-V or VT-x

Figure 1: Current x86 Virtualization Architecture

Major x86 processor manufacturers, such as AMD [5], [6] and Intel [7], [8] have developed proprietary virtualization techniques. The Virtual Machine Monitor (VMM) is the heart of the virtualization architecture. The VMM, which is a thin software layer between the VMs and real hardware, manages the VMs and the resources. The VMM abstracts all resources and exports a virtual copy to the VMs. In current architectures, the VMM only exports a virtual copy of the system's CPU and allocates memory to the VMs. Furthermore, the VMM does not provide virtual I/O. VM0 is a special VM that is trusted by the VMM. It emulates I/O and services I/O requests from all the VMs. VM0 also serves as a management console for the administrative users to create, modify and destroy VMs. The VMM accepts administrative commands only from VM0. Finally, VM_1-VM_n are general purpose VMs. These VMs use generic I/O drivers as they are supported by most operating systems. All I/O requests are trapped and redirected to VM0, by the VMM.

Several performance and security issues have been identified in this virtualization architecture [9], [10], [11], [12]. These issues arise from the fact that the current virtualization architecture depends on VM0 to provide critical I/O services.

Following is a summary of key security and performance issues:

- A VM implicitly trusts its host environment. If the host is compromised or is malicious, modifications to a VM's memory or its I/O communication channels can neither be detected nor avoided.
    - Ormandy [10] demonstrated several flaws in I/O emulation software used by popular VMM vendors. According to a recent bug report published by Secunia.com, an adversary could gain root privileges in VM0 by exploiting buffer overflow vulnerabilities in I/O emulation software [12].
- Since memory is shared by all VMs, allocation and de-allocation of memory can leak sensitive information.
    - The current architecture does not protect against information leakage in the event of failure. For example, VMMs in Xen [13], [14] and other virtualization platforms allow a VM to be suspended and resumed later. The memory and processor states of a suspended VM are stored in the VM0 file system. This allows an adversary, who controls VM0, to easily extract/modify sensitive information from any VM without being detected.

All VMs depend on a single controller called VM0 for critical I/O services. A compromised VM0 could either result in denial of service, loss of sensitive information or system compromise. Similarly, a compromised VM (not the privileged VM0) could result in denial of service as all VMs share common I/O channels.

- Frequent context switching between VM’s degrades system performance.
  - When a VM is scheduled out, its virtual processor state is stored in a large data structure (about 4K). Karger [9] shows that this could cause a serious performance problem if VM’s frequently switched the 4K of state information back and forth. Since the current architecture depends on a virtual machine (VM0) for I/O services, it has to be scheduled in and out frequently.
- The I/O throughput is constrained by the processing capacity of VM0.

2.2 Project Outcomes

We prototyped HAVEN using FPGA based secure co-processing to address the limitations of current virtualization technologies. Specifically, HAVEN:

- Increased reliability via a hardware-assisted virtual I/O subsystem for each VM.
- Improved performance by minimizing context switches back to the controller VM0 and by using a hardware virtual I/O manager.
- Improved security by protecting storage and communication channels using FPGA-assisted encryption and authentication.

The high assurance virtualization platform will enable:

- Use of virtualization in mission critical and high assurance applications.
- High assurance/high performance computing platform that provides application level compartmentalization.

Development of a high assurance Virtual Machine Monitor (VMM) with a micro-kernel implemented in hardware.

2.3 Vulnerabilities that HAVEN Targets and their Solutions

The HAVEN engine targets two key security vulnerabilities.

1. VM0 (Domain 0) manages virtualization of all I/O subsystems and is the single point of failure. An attacker that takes control of VM0 can covertly observe on other VM activities and launch man-in-the-middle attacks. Furthermore, since VM0 schedules in and out, it impacts overall system performance.
2. The memory states of all VMs are accessible to all VMs. When a VM is scheduled out, a 4K size state is stored in VM0's memory space. The state of a VM can be accessed directly by VM0 or indirectly by other VMs.

The memory isolation block deals with the first vulnerability. It contains mechanisms to fully encrypt (using separate keys) the memory spaces of individual guest operating system instances with the associated key-management infrastructure. The keys used to encrypt the memory spaces are neither available to the host nor the guest operating systems, thwarting man-in-the-middle attacks. Furthermore, when a VM is scheduled out, its 4K size state will contain the encrypted text. Additionally, hardware assisted virtual I/O devices will reduce the need to schedule VM0, improving overall system performace and minimizing the possibility of cross-contamination of data between guest VMs.

3 High Assurance Virtualization ENgine (HAVEN)

We designed an architecture that improves overall performance, reliability and security in virtualization. Sailer et al [14] identified important goals a medium assurance VMM should meet. These include strong isolation, controlled sharing, integrity, accountability and secure services. Our architecture, HAVEN, depicted in Figure 2, is designed with these goals in mind.

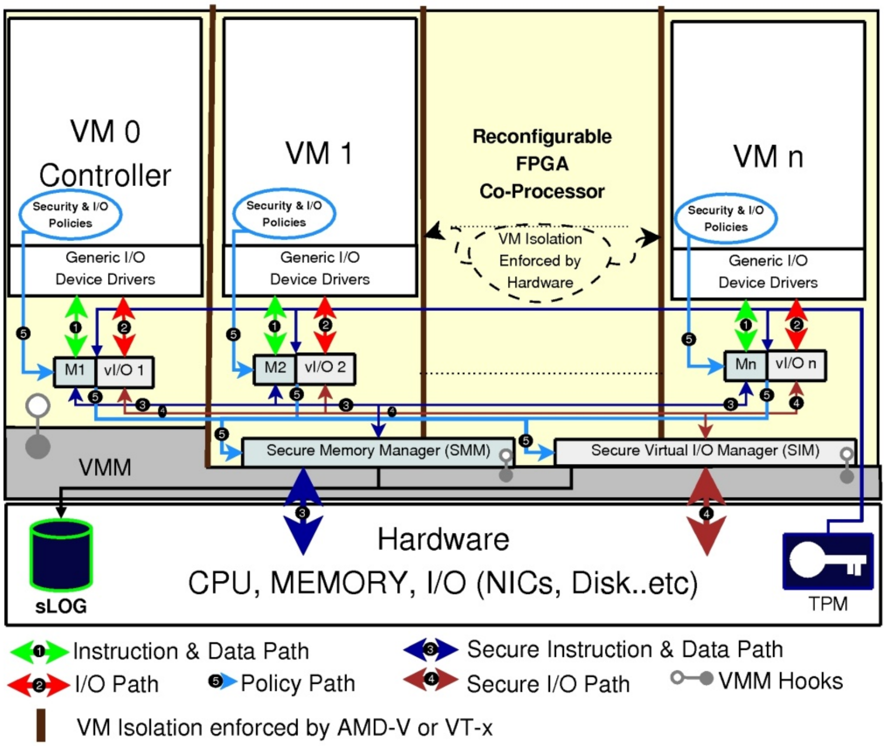

VM 0
Controller
Security & I/O Policies
Generic I/O Device Drivers
VM 1
Security & I/O Policies
Generic I/O Device Drivers
Reconfigurable FPGA Co-Processor
VM Isolation Enforced by Hardware
VM n
Security & I/O Policies
Generic I/O Device Drivers
M1
vI/O 1
M2
vI/O 2
Mn
vI/O n
VMM
Secure Memory Manager (SMM)
Secure Virtual I/O Manager (SIM)
Hardware
CPU, MEMORY, I/O (NICs, Disk..etc)
sLOG
TPM
Instruction & Data Path
I/O Path
Policy Path
Secure Instruction & Data Path
Secure I/O Path
VMM Hooks
VM Isolation enforced by AMD-V or VT-x

Figure 2: Proposed HAVEN Virtualization Support for x86 Architectures

3.1 Overview of HAVEN components and subcomponents

Figure 3 shows the HAVEN components and their association with a commodity computer architecture. A commodity computer mainly consists of the CPU, the north bridge and the south bridge. The north bridge is the primary communication hub in a commodity computer and the south bridge provides connectivity to in built I/O devices. The north bridge communicates with the CPU and passes on the I/O and memory requests to and from the CPU to the PCI/PCIe bus and main memory. Different components hang off the PCI/PCIe bus. The south bridge also attached to the north bridge throught the PCIe bus, which communitcates with the different I/O components as shown in Figure 3.

The HAVEN architecture consists of Secure I/O Manager (SIM) and Secure Memory Manager (SMM). The SMM is connected to a RAM chip which is used as the system memory. It is the primary memory of the HAVEN architecture. Traditionally, the RAM on the right side of the north bridge is the primary memory. However, in the HAVEN architecture it is used only during the boot up cycle. Once the system boots up, all memory transcations go over the PCIe bus to the SMM. The SMM manages the encryption and decryption of the data.

The SIM is prototyped for the a network interface card(NIC). The NIC is traditionally connected to the south bridge. However, the SIM is connected directly to the north bridge via the PCI bus. To understand the HAVEN architecture in more detail lets go over the various steps a commodity computer goes through at boot up time as shown in Figure 3.

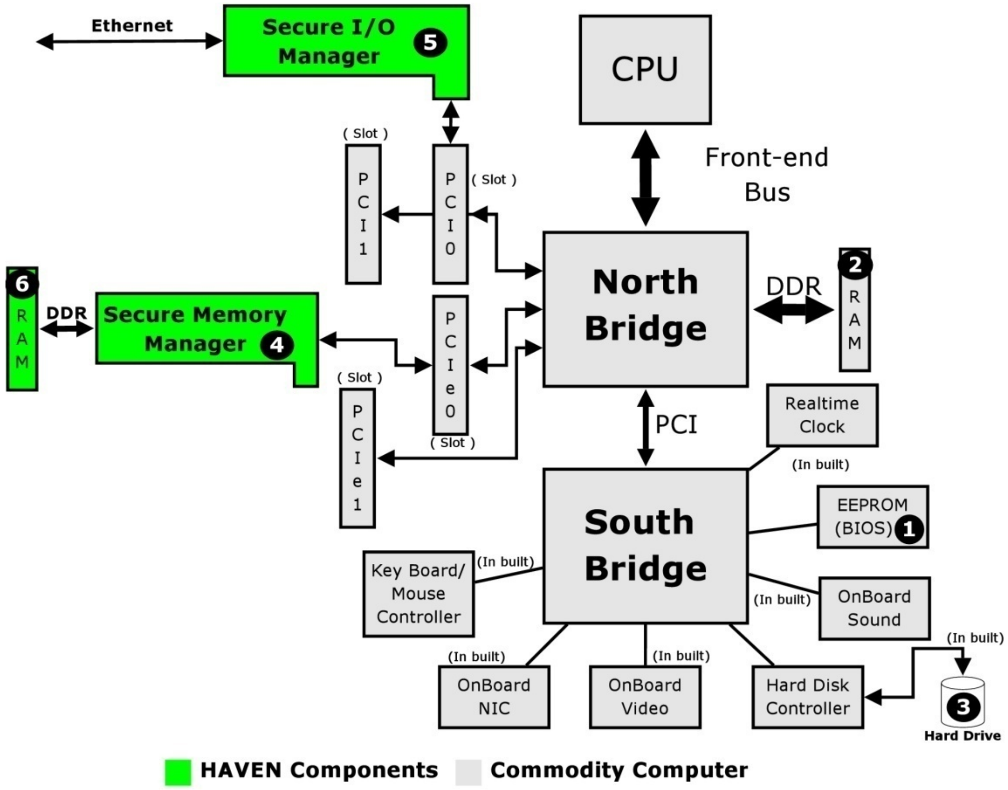


Figure 3: Layout of a Commodity Computer including HAVEN Components

1) The CPU exits the reset state and executes the BIOS from the EEPROM. The BIOS initializes the North Bridge and Main Memory and copies itself into the Main memory.
2) The BIOS starts to execute from Main memory and initializes the South Bridge and I/O devices connected to South Bridge.

3) The BIOS executes the OS loader from the hard drive. The OS loader load XEN VMM into the main memory.
4) XEN's VMM initializes the PCIe and PCI devices, including SMM and SIM, connected to the north bridge and loads VM0's kernel image into main memory and executes it.
5) VM0 scans SIM for virtual I/O devices and allocates them to guest VM's
6) Xen's VMM uses memory behind SMM for VM's created by VM0.

Figure 4 shows the HAVEN components assembled with a commodity computer motherboard.

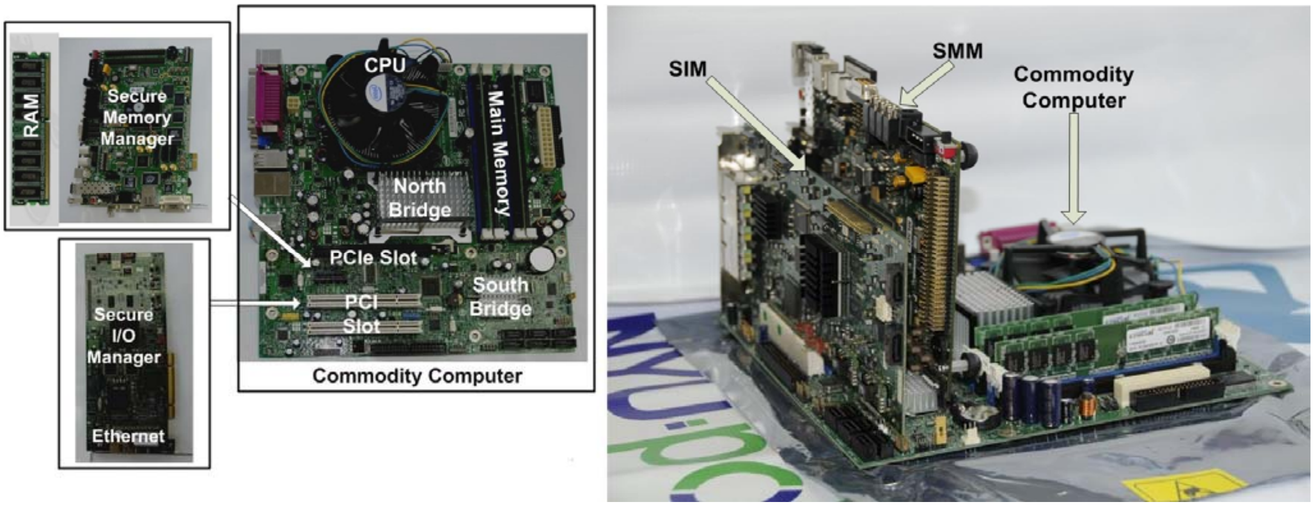


Figure 4: HAVEN Components Assembled with a Commodity Computer Motherboard

3.2 Secure Virtual I/O Manager (SIM)

I/O virtualization provides a separate virtual device for each running VM. Virtualizing an I/O device essentially translates into the problem of scheduling a resource between multiple VMs. This requires multiplexing, de-multiplexing, and scheduling the resources. In addition, requests and data for individual VMs are buffered when the physical I/O device is servicing another request.

As far as the CPU is concerned, virtualizing an I/O device means that the CPU should be able to detect multiple I/O devices of the same type on the PCI bus during boot up. For example, if there is a single physical ethernet network interface card (NIC), the CPU should be able to see multiple NICs when the machine boots up. This requires that both the PCI bus as well as the NICs (and I/O devices in general) be virtualized.

3.2.1 Virtualizing the PCI Bus

PCI is an interface bus protocol that connects I/O devices with the CPU. It is a synchronized bus running on either a 33 MHz or a 66 MHz clock. In this prototype, we support the 33 MHz clock. To understand the PCI bus protocol, let us look at the different PCI bus signals:

- CLK (Input) provides timing reference for all signals and data transfers.
- RSTn (Input) is an active low asynchronous reset signal.
- AD[31:0] (Bidirectional) is a 32-bit multiplexed bus. This bus is used as an address bus at the start of a PCI transaction (address phase), and it contains the address of the target devices. All other times it is used as a data bus.
- CBE[3:0] (Input) is a 4-bit multiplexed bus. During the address phase CBE is used as command bus and all other times it used as byte enable.
- FRAMEn (Input) is an active low signal asserted to indicate the start of a new transaction. The clock cycle following FRAMEn is the address phase.
- IRDYn (Input) is an active low signal asserted to indicate that the master is ready to accept data or has driven valid data into the AD bus.
- TRDYn (Output) is an active low signal asserted by the target to indicate that it has either latched the data in the AD bus (for write transaction) or has driven valid data into the AD bus (for read transactions).
- DEVSELn (Output) is an active low signal asserted by the target device to indicate that it intends to claim the current transaction.

Typical PCI bus transactions involve a master and a target. The master initiates the transaction, and the target replies to the request. Common types of transactions are I/O read and write (0x2 and 0x3), memory read and write (0x6 and 0x7), and configuration read and write (0xA and 0xB). The sequence and control flow is identical to all transactions with the only difference being the command data. For readability purposes, “0x” is affixed in the front of all hexadecimal values and “b” appended to binary values throughout this report.

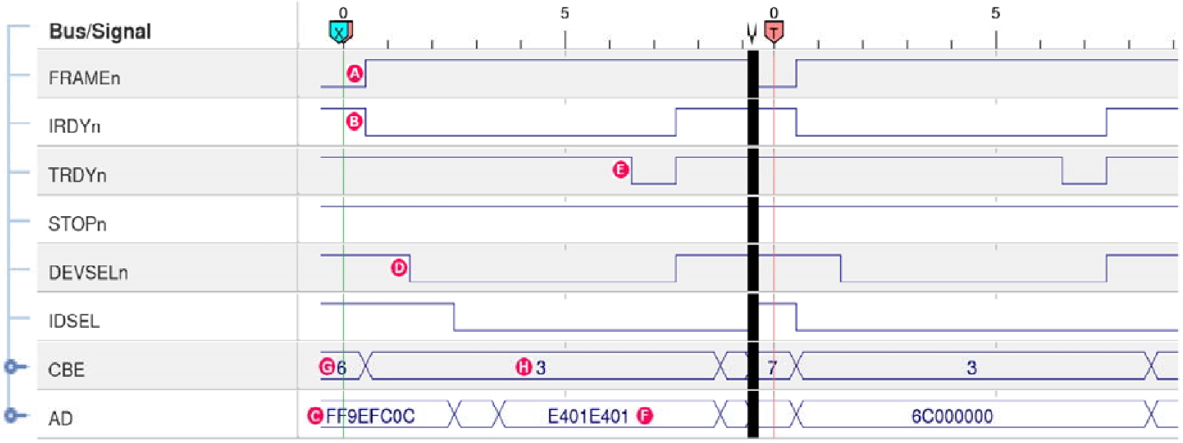


Figure 5: Memory Read Transaction

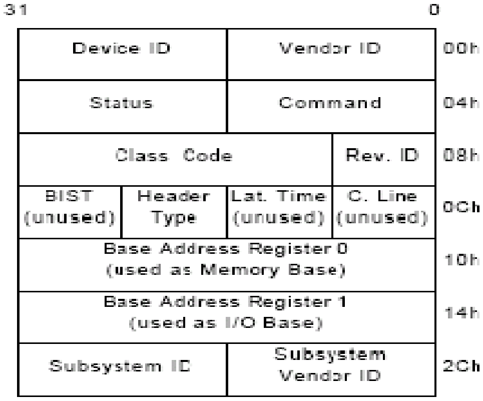
Figure 5 illustrates a Memory Read transaction. The transaction begins when the master asserts (set to low) FRAMEn (A). This drives target address onto the AD bus (C), and the command to be executed onto the CBE bus (G). In the case of a memory read, the command data value on the CBE bus is set to "0x6." At the end of the first clock cycle, the master asserts IRDYn (B) to indicate its readiness to accept data. The data on the CBE bus (H) is then changed to indicate the "byte enable" data the master is expecting. The value indicates which of the four bytes of the AD bus the master expects to read. The data on the CBE bus represents a negative mask of the data expected. Therefore, the value "0011b" at (H) indicates that the master will read the two high order bytes (16-bits) from the AD bus.

During the address phase, the target device latches the contents of the AD and CBE buses. The latched address is compared with the base address registers and if they match, the device asserts DEVSELn (D) and proceeds to execute the command latched during the address phase. The latched AD bus indicates what memory address to read. Once the read operation is complete, the target asserts TRDYn (E).

Putting this all together, a read (command 0x6 on CBE) from memory address 0xFF9EFC0C is requested (asserting FRAMEn) by the master at time X. At time X+1, the master states its readiness to accept (asserting IRDYn) the highest two bytes of data (0011b on CBE). Right afterwards, the target device saves the address (value on AD) and byte mask (value on CBE) and compares the saved address value with the base address register. Since they match, the target claims the current transaction by asserting DEVSELn at time X+2. It then performs the requested read and writes the data value (0xE401E401) onto the AD bus. Once it has completed the read, it asserts TRDYn (time X+6.)

3.2.2 Virtualizing a PCI Device

The previous section presented a view of the PCI bus protocol, but there is still a need to virtualize a PCI device controller, such that the CPU will recognize it. The PCI architecture supports two ways in which different configurations can share a single physical PCI connector: PCI Multi-Function Device and PCI-to-PCI Bridge. The PCI Multi-Function Device specification allows a single PCI device to host up to eight (8) different functions. Examples of this would be a Multi-Function card with a parallel port, USB port, and serial port. From the software point-of-view, a Multi-Function device presents each function as if it were a completely separate PCI device.



31			0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Rev. ID	08h
BIST (unused)	Header Type	Lat. Time (unused)	C. Line (unused)	0Ch
Base Address Register 0 (used as Memory Base)				10h
Base Address Register 1 (used as I/O Base)				14h
Subsystem ID		Subsystem Vendor ID		2Ch

Figure 6: Configuration Header

Every PCI device maintains the configuration header shown in Figure 6. The configuration header is used to manage the device and establish the memory and I/O address space for the device. Many of the fields in the configuration header are static and do not change once set by the vendor. It is the configuration header which is used to differentiate the functions within the PCI card and establishes the I/O access to a specific function on the PCI card.

To virtualize the PCI controller, we used the multi-function device feature and maintained a virtual configuration header for each virtualized function. The virtual configuration headers were then multiplexed on the FPGA in order to give the appearance of separate physical devices.

3.2.3 Implementation of Virtual PCI Conroller

By maintaining separate configuration headers for each virtual device on a device, we can multiplex the physical I/O into separate virtual devices. The virtual configuration headers will be used to manage the state of the virtual devices and access to the virtual devices as though they are physically distinct.

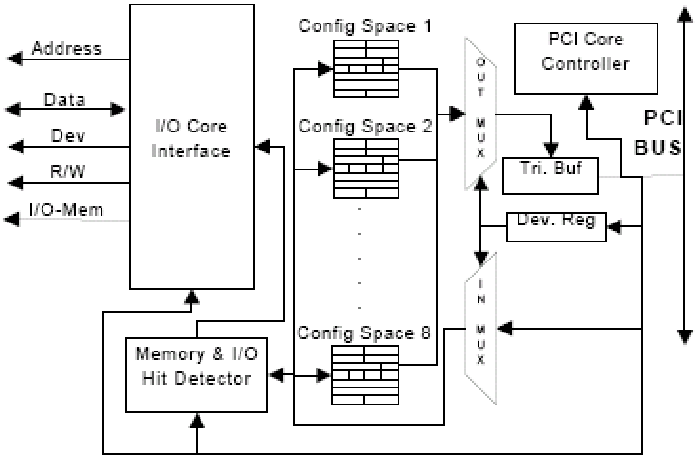


Figure 7: High Level PCI Virtualization Architecture

Figure 7 is the high level PCI virtualization architecture. Information such as the base addresses are set at boot time by the OS and are exclusive to the device. Since we are virtualizing the PCI controller by constructing multiple configuration headers for a single I/O device, the address space is limited to eight devices. Although the second option does not suffer from this limitation, the underlying architecture is more complicated and will be investigated in the future.

The virtual PCI core was implemented and tested on an AVNET Spartan-2 FPGA development board. Xilinx ChipScope was used to verify the correct operation.

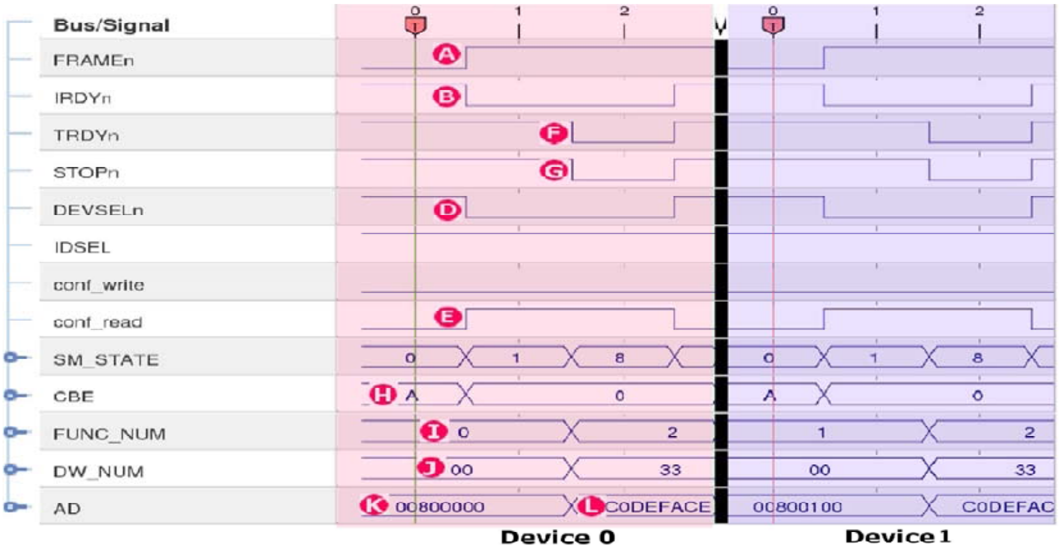


Figure 8: PCI Device Configuration Read

Figure 8 shows the reading of configuration headers of virtualized PCI devices. The virtual device number for a transaction is shown as FUNC NUM. The PCI Virtualization Controller latches the function number and uses it to select an appropriate virtual PCI device. The signal conf_read (E) is an internal signal used to track each transaction. Signal DW_NUM (J) indicates the configuration register that is being read. SM_STATE shows the internal state of the PCI Virtualization Controller on each clock cycle.

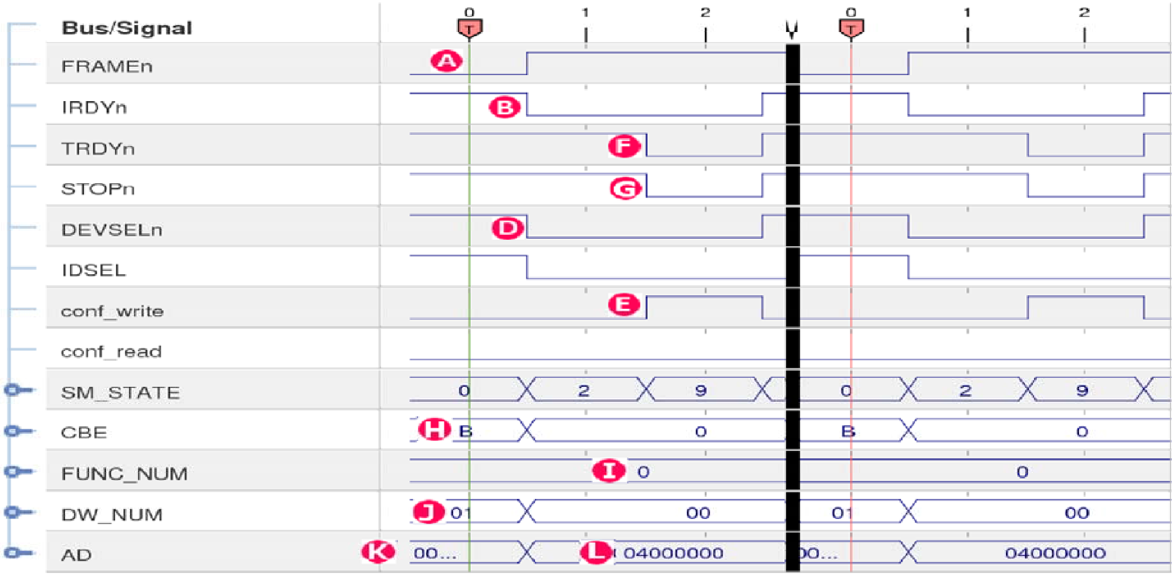


Figure 9: PCI Device Configuration Write

Figure 9 shows the writing of data into a configuration header of a virtualized PCI device. Once again, FUNC_NUM is the virtual device number and the PCI Virtualization Controller uses this FUNC_NUM to select the appropriate virtual PCI device. The signal conf_ write (E) is used to track each write transaction. Signal DW_NUM (J) indicates the configuration register that is being read, and SM_STATE shows the PCI Virtualization Controller state.

3.2.4 Virtualizing an I/O Device

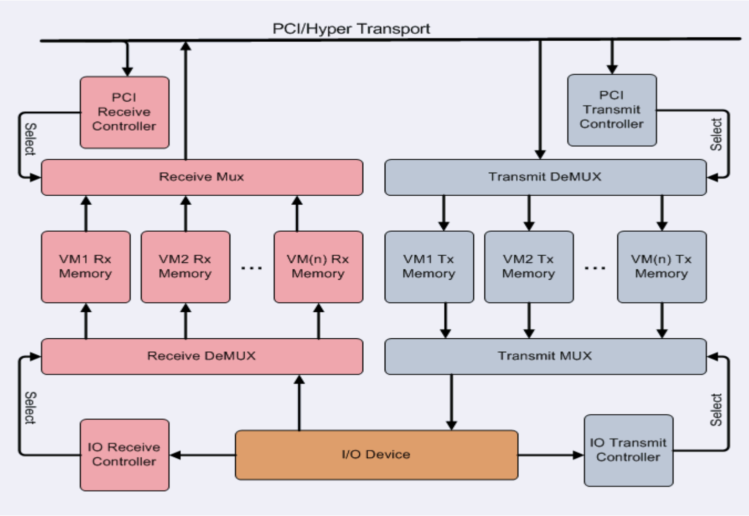


Figure 10: General I/O Virtualization Architecture

The I/O device Virtualization Controller interacts with the I/O device on one side and the virtual PCI controller on the other side. Figure 10 outlines the general architecture used for I/O virtualization. The virtualization architecture has a transmit part (left) and a receive part (right). The I/O device is at the bottom of this figure and the virtual PCI controller is at the top. The receive and transmit sides have their own multiplexers (MUX), de-multiplexers (DeMUX), controllers, and buffers for each VM.

Receive datapath: The I/O device is the source and the VM is the destination. The I/O device sends data and destination information. Based on this information, the I/O receive controller generates the appropriate select lines for the receive DeMUX. The data is moved from the I/O device to the appropriate VM RX memory. The VM RX Memory stores the data and relevant information while the CPU is busy, waiting for the VM to fetch the data from its memory.

When a VM is scheduled to run on the CPU, it checks to see if there is any data in its memory (this can be done by using flags or interrupts). If there is valid data in memory, then the VM sends the appropriate address via the PCI bus to virtual PCI controller, which forwards the request to the receive part of the I/O Virtualization Controller. The request is processed by the I/O Virtualization Controller which then generates the correct select lines for the receive mux. The VM communicates by using the given protocol and retrieves the data from its memory.

Transmit datapath: Here the VM is the source and the I/O device is the destination. When a VM is scheduled to run and is ready to send data to the I/O device, it sends the appropriate I/O address via the PCI bus to the virtual PCI controller to the virtual PCI device. The request is processed by the PCI transmit controller and generates correct select lines for the transmit DeMUX. The VM communicates by using the given protocol and transmits the data to its memory.

When the I/O device is ready to receive data, the I/O transmit controller selects one of the VMs out of the ones which are ready to transmit by using a round-robin-based scheduling algorithm. The I/O transmit controller uses the appropriate protocol to communicate with the I/O device and send the data out of the memory.

The I/O device transparently receives and transmits data. For example consider the ethernet device. It receives and transmits valid packets and is not concerned with the contents of the packet. The I/O virtualization is transparent to the I/O device since the appropriate protocol is being followed.

Though the PCI bus is controlled by the CPU, the CPU is also unaffected by I/O virtualization because it detects multiple independent I/O devices. Thus, as far as the CPU is concerned, it is exchanging data with a distinct I/O device.

The drawback of this architecture is that I/O devices will be busier with virtualization support. There is a higher chance that the I/O device will be unavailable when a VM is running on the CPU and is trying to access the I/O device. This can be resolved by increasing the size of the buffers dedicated to individual VMs, but remains to be implemented and verified.

3.2.5 Implementing a Controller to Virtualize the Ethernet MAC

An ethernet device contains the actual physical TX/RX device and a Tri-mode Ethernet Media Access Controller (TEMAC) core that communicates with the physical device on one side and relays data between the physical device and the CPU via the PCI bus on the other side. The Virtualization Controller is implemented between the virtual PCI controller and the TEMAC.

3.2.5.1 *Virtualizing the packet receive function*

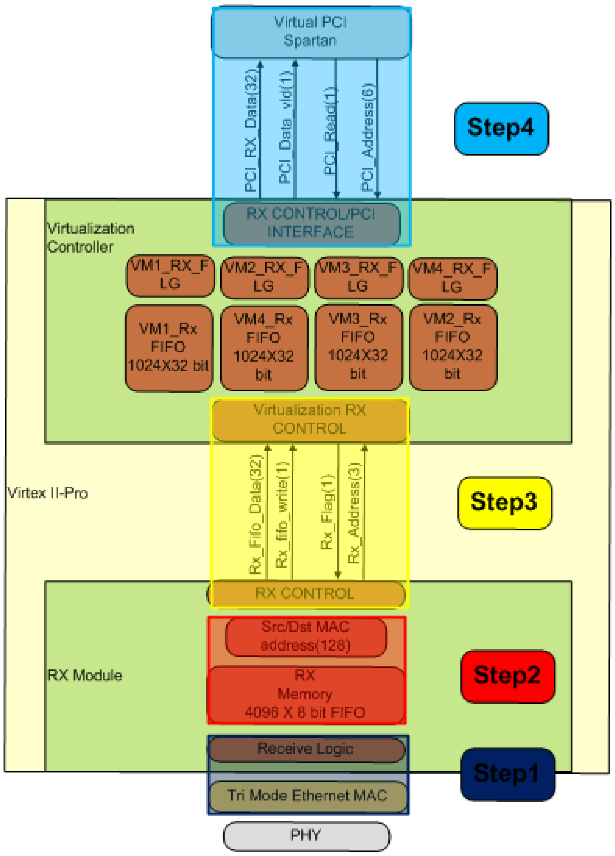


Figure 11: Virtualizing the Ethernet Controller

Figure 11 shows the receive module of the Ethernet Virtualization Controller. The RX module sets up the protocol with the TEMAC device and then moves the data to the Virtualization Controller. The virtual PCI controller (top most box of figure) reads the data from the VM memory via the PCI bus. The VM checks the status of its flag before reading the data. The meaning and purpose of each step in Figure 11 is explained below:

Step 1: When a packet comes into the physical device, the device passes the packet to the TEMAC, which in turn passes it on to the RX Module. The receive logic shown in Figure 11 communicates with the TEMAC and stores the packet into the RX Memory. The RX memory can hold four kilobytes (the size of a packet). In this case, RX memory is a first-in, first-out (FIFO) memory module.

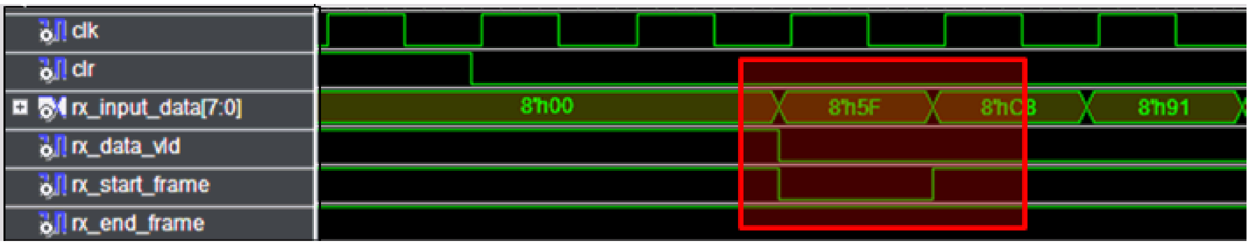


Figure 12: Start of New Frame (Step 1)

Figure 12 shows the exact protocol between the TEMAC and Receive Logic. The TEMAC essentially has 4 input signals: RX Data, RX Data Valid, RX Start Frame, and RX End Frame. All signals are active low; thus, when a new frame is being received, the TEMAC sends a signal by pulling the RX Start Frame to zero for one clock cycle. The data is transmitted in blocks of 8 bits along with an active low RX Data Valid signal. Figure 12 shows that the first few frames are 5F, C8, and 91.

Data is transmitted and stored in the RX Memory until the last frame is received. TEMAC sends a signal by pulling the RX End Frame to zero for one clock cycle as shown in Figure 13.

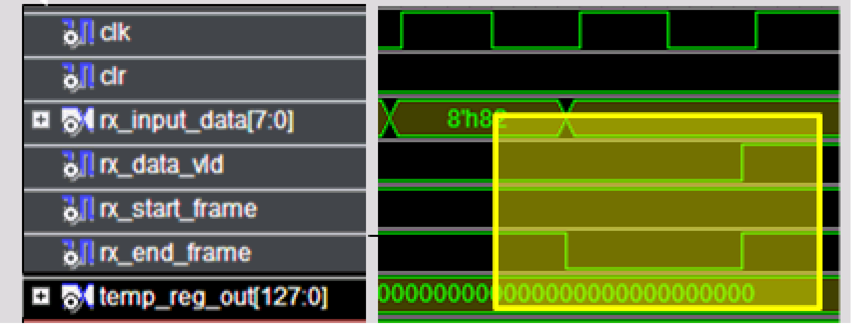
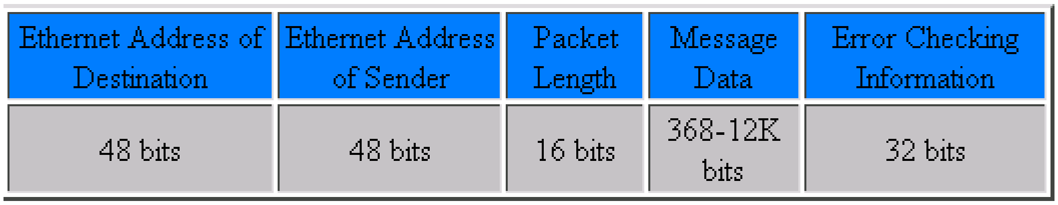


Figure 13: End of Frame (Step 1)

Step 2: Once the packet is received, the first 128 bits of the packet are moved to a temporary register (Src/Dst MAC address) as shown in Figure 11. Data is moved in 32 bit blocks. The following figure shows the structure of an Ethernet Frame.



Ethernet Address of Destination	Ethernet Address of Sender	Packet Length	Message Data	Error Checking Information
48 bits	48 bits	16 bits	368-12K bits	32 bits

http://www.cs.williams.edu/~tom/courses/105/outlines/CS105_227.html

Figure 14: Structure of an Ethernet Frame

The first 48 bits of the Ethernet frame give the destination MAC address, which is compared to the MAC addresses of all the VMs. If a match is found, then it is kept for further processing, otherwise, it is discarded. The first 128 bits are moved to the temporary register so that the destination, source address, and packet length can be checked. Figure 15 shows the simulation of this step. Since the data is moved in 32 bit blocks, it takes 4 cycles to move 128 bits to the temporary register.

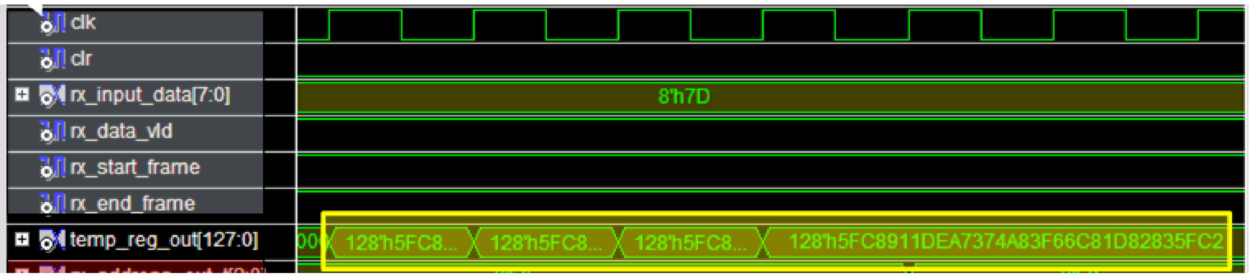


Figure 15: First 128 Bits moved to Temporary Register (Step 2)

Step 3: Once the MAC address has been identified and matched, the corresponding VM memory is checked; in other words, the status of the VM flag is checked. If the flag indicates that the memory is ready to receive a new packet, then the packet is sent to the VM FIFO from the RX Memory.

The simulation in Figure 16 shows the protocol between the RX Control and Virtualization RX Control. The following diagram illustrates the protocol:

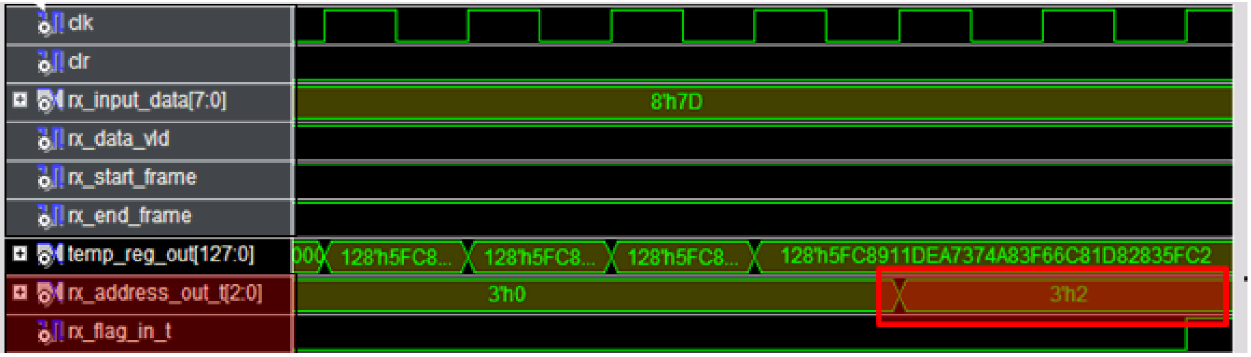


Figure 16: Address sent to Virtualization Controller (Step 3)

- RX address gets the address of VM FIFO associated with the owner of the packet. In this case, it is found that the MAC address belongs to VM2.
  - If that VM's FIFO is empty, then the RX flag signal is sent by the Virtualization RX Control.
  - If the VM2's FIFO is not empty, then no signal is sent.
- RX module waits for 10 cycles and either:
  - Discards the packet if no RX flag signal is received or
  - Sends data with an RX FIFO write signal if the RX flag signal is received. After the entire packet is sent, the VM Flag is set, which informs that VM that there is a valid packet in the VM FIFO.

Figure 17 shows that the VM2's FIFO memory was empty and thus sent an RX flag signal. As soon as the signal is received, the data is transmitted to the VM2's FIFO along with an RX FIFO write signal.

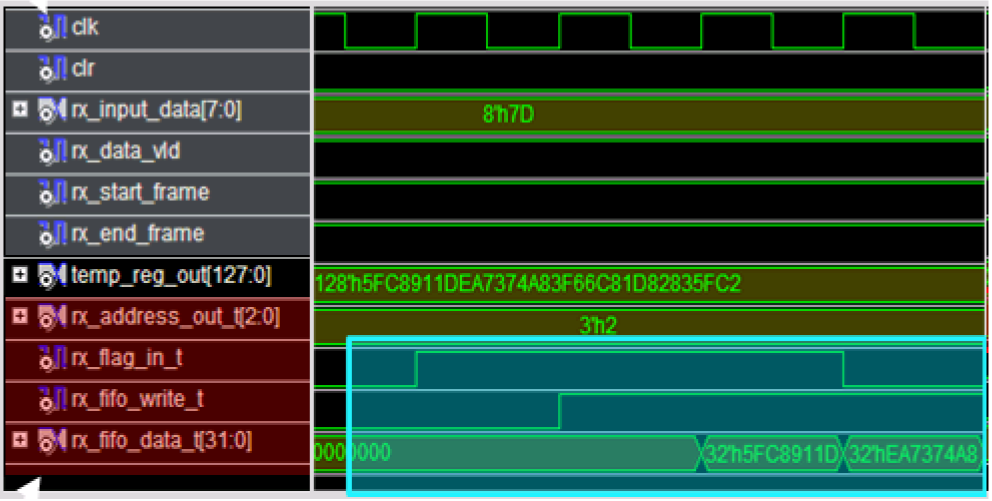


Figure 17: Data sent after Flag Status is Checked (Step 3)

Figure 18 shows the end of the frame; i.e. the entire packet is sent to the VM2's FIFO and the RX FIFO write signal goes down to zero. Once the entire packet is sent, the VM2's flag is set so that the flag informs the virtual PCI controller that VM2's FIFO has a valid packet.

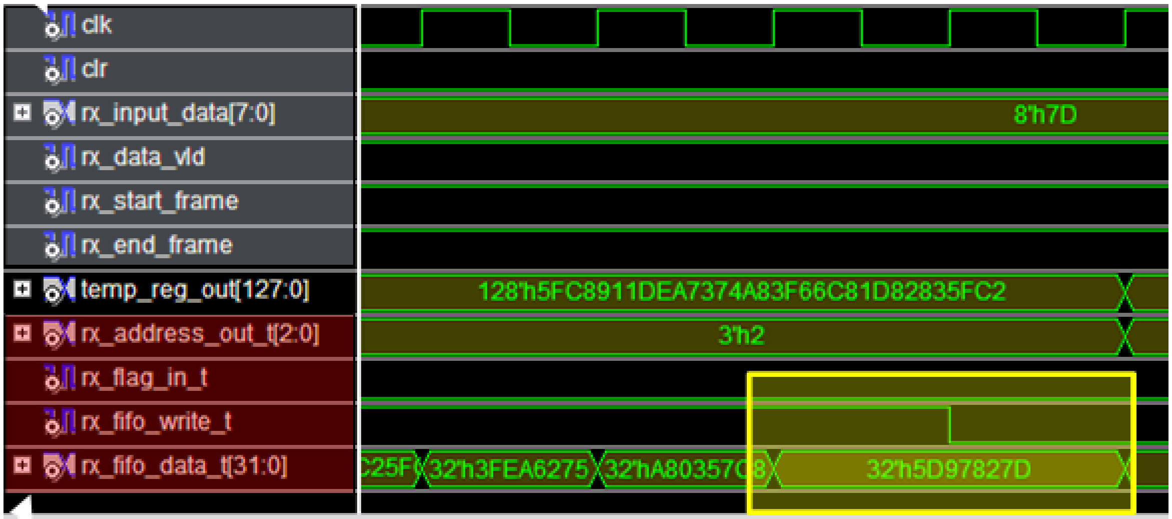


Figure 18: Last Few Bytes of Packet are moved to the Virtualization Controller (Step 3)

Step 4: Each VM interacts with the Ethernet Virtualization Module via the virtual PCI controller. The virtual PCI device sends an address which is six bits to the Virtualization Controller. The most significant three bits give the device number, i.e. the VM number. For example, a value "011b" specifies VM3. The least significant three bits give the operations. In the case of the receive module, the least significant three bits have value "011b." This is not shown in the simulations below; however, it is explained later in more detail.

When a VM is scheduled, it sends its FIFO address along with a read signal on the PCI bus. The virtual PCI controller reroutes the address (RX address) and read signal (RX read) to the Virtualization Controler which in turn routes it to the NIC. The RX Control/PCI Interface checks the VM Flag corresponding to the address. If there is data in the VM's FIFO, it is sent in 32 bit blocks with an associated valid signal over the PCI bus. Figure 19 shows the communication between the virtual PCI controller and Virtualization Controller.

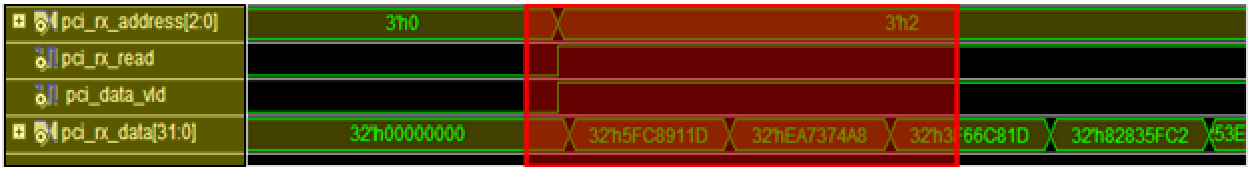


Figure 19: PCI Reads Data from VM2's Memory

As shown in Figure 19, the PCI sends an address with value "0x2" to the Virtualization Controller along with a PCI read signal. Since there is valid data in VM2's FIFO, the data is sent to the PCI along with a data valid signal. When the last frame is sent, the data valid signal goes low, as shown in Figure 20.

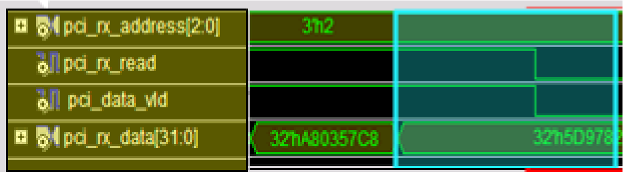


Figure 20: Last Few Bytes sent to VM

PCI Address Lines: As mentioned earlier, the virtual PCI sends six address bits to the Virtualization Controller. The most significant three bits provide the VM number. The least significant three bits give the operation. The following table explains the operations in detail.

Table 1: PCI Address Operations

Address	Operation
XXX000	Read most significant 32 bits of MAC address
XXX001	Read least significant 16 bits of MAC address
XXX010	Write to a transmit FIFO
XXX011	Read from Receive FIFO
XXX100	Finished writing to transmit FIFO i.e. set Flag

For example, address “010 000” signifies “read the most significant 32 bits” of VM2’s MAC address. This has to be done because the MAC address of each VM is hardcoded in the Ethernet Virtualization Controller. Thus, Ethernet Virtualization must accommodate a method for communicating the MAC address with the VM’s operating system. The following table gives the values of the PCI addresses for each VM.

Table 2: PCI Address Operation in Detail

PCI Address (in hex)	Meaning
00	Read most significant 32 bits of MAC : VM1
08	Read most significant 32 bits of MAC : VM2
10	Read most significant 32 bits of MAC : VM3
18	Read most significant 32 bits of MAC : VM4
01	Read least significant 16 bits of MAC : VM1
09	Read least significant 16 bits of MAC : VM2
11	Read least significant 16 bits of MAC : VM3
19	Read least significant 16 bits of MAC : VM4
02	Write to Transmit FIFO : VM1
0A	Write to Transmit FIFO : VM2
12	Write to Transmit FIFO : VM3
1A	Write to Transmit FIFO : VM4
03	Read from Receive FIFO : VM1
0B	Read from Receive FIFO : VM2
13	Read from Receive FIFO : VM3
1B	Read from Receive FIFO : VM4
04	Set Flag : VM1
0C	Set Flag : VM2
14	Set Flag : VM3
1C	Set Flag : VM4

3.2.5.2 Virtualizing the packet transmit function

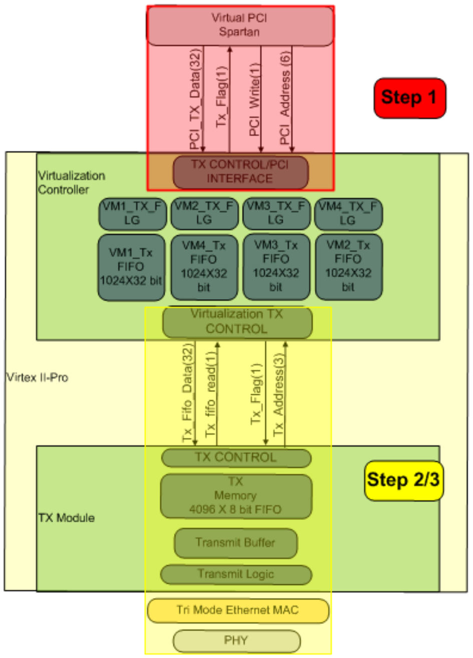


Figure 21: Ethernet Virtualization: Transmit Controller

Figure 21 shows the transmit module of the Virtualization Controller. The TX Module deals with setting up protocol with the TEMAC and fetching the data from the Virtualization Controller. The virtual PCI controller writes the data to the VM memory via the PCI bus. After the data/packet is completely written into the VM memory, the VM sets its flag to communicate that the packet is ready for transfer. Before transmitting a packet, the TX Module checks the status of the flag and then chooses one of the VM whose flag is then set. The flag is unset when the entire packet is sent to TEMAC. The meaning and purpose of each step in Figure 21 is explained below:

Step 1: When a VM is ready to transmit a packet, it sends a PCI Address which corresponds to the transmit packet and its VM number. Once the address is received by the Virtualization Controller, the status of the flag is sent back to the PCI device. If the flag is unset, then the VM sends data in 32 bit blocks along with a PCI TX write signal. Once the VM sends the complete packet, it sets the flag by sending an address corresponding to 'set flag' and its VM number. This is shown in the figure below.

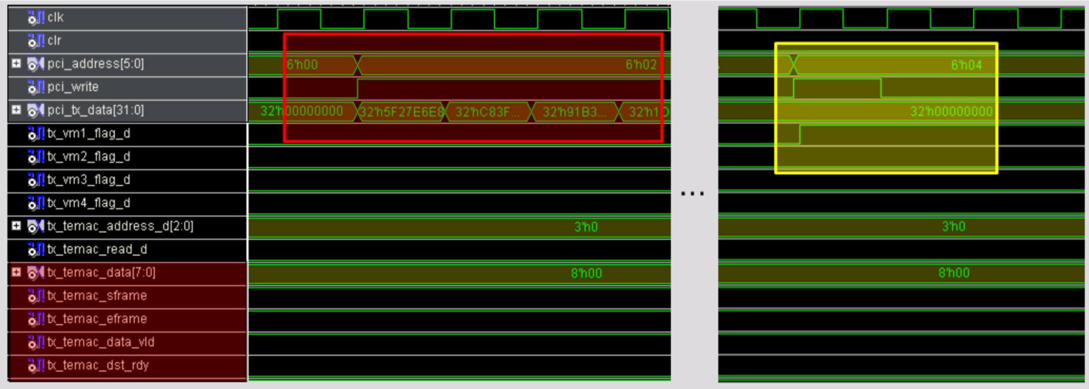


Figure 22: Data moved from PCI to VM Memory

Figure 22 shows that VM 1 is sending data to its memory in the Virtualization Module via the PCI bus and then through the virtual PCI controller. An address "0x02" is sent to the Virtualization Module. Since VM1 flag is unset, meaning "0b" data is transmitted to the VM memory in 32 bit blocks. Once the entire packet is sent to the VM memory, the VM1 sends an address "0x04" to set its flag along with a PCI write signal.

Step 2/Step 3: The TX Controller in the TX Module constantly monitors the status of the flags in the Virtualization Controller. As soon as a flag is set, it tries to send the packet to the TEMAC. The TX Controller sends an address along with a read signal to the Virtualization Controller, which corresponds to a particular VM FIFO. The data is either moved from the VM memory to a temporary TX Memory in 8 bit blocks or sent out directly, depending on the status of the TEMAC. If the TEMAC is ready to transmit a packet, then the data is sent to the TEMAC directly, otherwise, it is temporarily stored in the TX Memory. The figure below shows the exact protocol used while sending a packet.

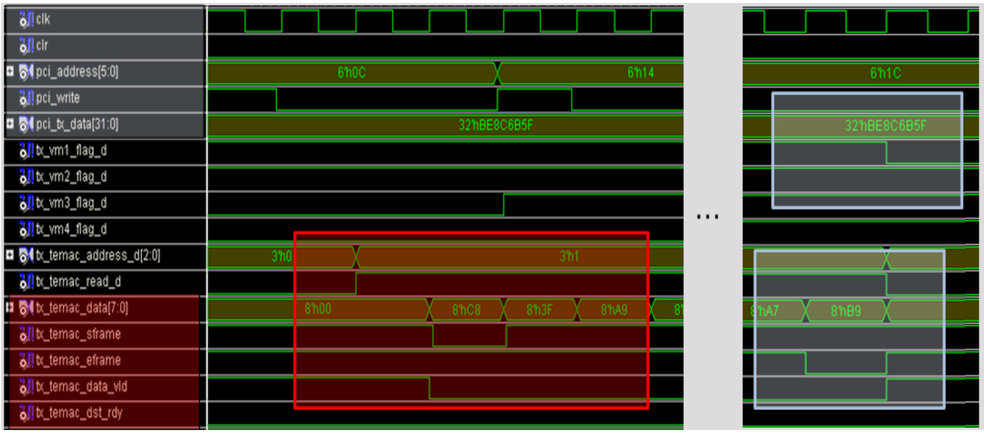


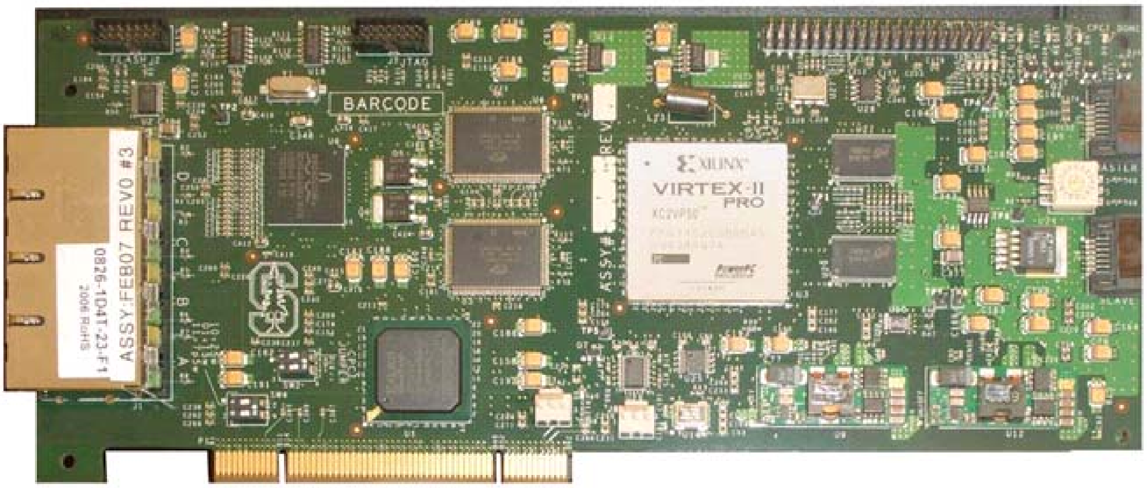
Figure 23: Packet is sent to TEMAC from VM Memory

Since the VM1 flag is set, the TX Module sends an address value “1” along with a read signal to the Virtualization Controller. The data is transmitted in 8 bit blocks. Since the TX TEMAC destination ready is low, the data is directly sent to TEMAC without temporarily storing the data in the TX Memory.

TEMAC basically has 5 input/output signals: TX TEMAC start frame (input), TX TEMAC data (input), TX TEMAC end frame (input), TX TEMAC data valid (input), and TX TEMAC destination ready (ouput). All signals are active low. When sending a packet to TEMAC for transmission, first the status of the destination is checked. If the TX TEMAC destination ready signal is low, then the packet is sent. To send a packet, a TX TEMAC start frame signal is sent along with TX TEMAC data valid and the first 8 bits of the TX TEMAC data. Once the entire packet except the last 8 bits are sent, a TX TEMAC end frame signal is sent. As soon as the entire packet is sent, the VM flag is unset, as shown in Figure 23.

3.2.6 Ethernet Virtualization Implementation Results

Ethernet virtualization has been implemented and tested on the Virtex2P, device XC2VP50, on the NetFPGA development board. A picture of the NetFPGA development board is shown below.



http://www.netfpga.org/specs.php

Figure 24: NetFPGA Development Board

The number of cycles taken by the receive and transmit modules of the Ethernet Virtualization Module depends on the size of the packet. The TEMAC communicates data in one byte blocks. Assuming that packet size is 4 Kibibytes, it will take 4096 cycles to receive and transmit a packet.

For the receive module, there is an additional delay of 4 cycles to move data to the temporary register and an additional 1024 cycles to move the packet to a VM memory. The packet is sent to virtual PCI in 32 bit blocks; thus, it takes 1024 cycles. For the transmit module, it takes about 1024 cycles to move a packet from the VM to the Ethernet Virtualization Module in 32 bit blocks. If the TEMAC is not busy, the packet is sent immediately in 8 bit blocks, which take around 4096 cycles. Table 3 shows the device utilization of the Virtex2P FPGA.

Table 3: Device Utilization for Ethernet NIC Virtualization

Number of Slices	11167 out of 23616 (47%)
Number of Slice Flip Flops	7410 out of 47232 (15%)
Number of 4 input LUTs	20887 out of 47232 (44%)
Number used as logic	4419
Number used as shift registers	40
Number used as RAMs	16428
Number of IOs	160
Number of bonded IOBs	160 out of 692 (23%)
IOB Flip Flops	3
Number of BRAMs	14 out of 232 (6%)
Number of GCLKs	6 out of 16 (37%)
Number of DCMs	2 out of 8 (25%)

Although the the maximum frequency of the Ethernet Virtualization Module is 205.7 MHz, we operate it at 125 MHz since the TEMAC operates at 125MHz. Also, since the PCI architecture operates at 33MHz, it is a major bottle neck. In our implementation, PCI is the bottle neck, however if a faster I/O bus is used, the performance can be improved significantly.

3.2.7 Performance Improvements

Hardware assisted I/O virtualization reduces the need to schedule VM0, which is needed in the current architecture to perform all I/O operations. By not involving VM0 for I/O operations we save on expensive VM context switches (VMn to VMM and then VMM to VM0). To execute a context switch the CPU has to approximately swap 8 kilobytes (4 kilobytes per context switch) of data/instructions from the CPU to memory and vice-versa not including possible page faults. Therefore by removing VM0 from the virtual I/O subsystem, we save on at least two context switches per I/O request. Experimental results have shown that VMs perform exceptionally well using a hardware virtual I/O subsystem as compared to a software I/O subsystem [16].

3.2.8 Device Driver to Support Ethernet Virtualization

Modern x86 computers use the PCI bus to interconnect I/O devices with the CPU. x86 CPUs by themselves do not have the ability to communicate with PCI devices directly; a PCI host bridge and a device driver are required. A device driver for virtual Network Interface Cards (NIC) was developed under Linux. The PCI device driver provides the software hooks to communicate with SIM. The device driver does not provide virtualization.

3.2.8.1 A general device driver in Linux OS

A device driver is the interface between the Kernel and I/O device. Its primary purpose is to command the I/O device when OS transfers data or makes status queries. Typically an I/O device has some built-in memory which is mapped into the address space of the CPU for data transfer. This memory that is internal to the I/O device is mapped by the device driver during initialization as shown in Figure 25 (a).

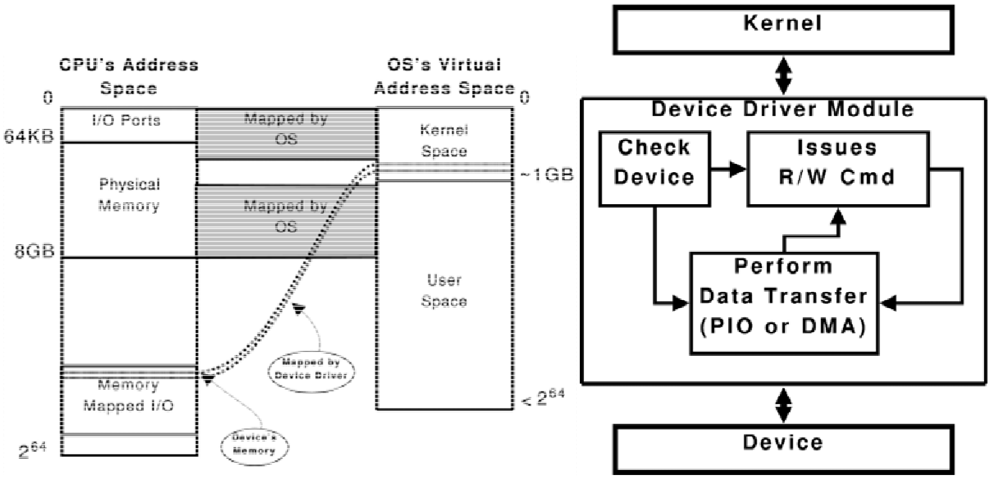


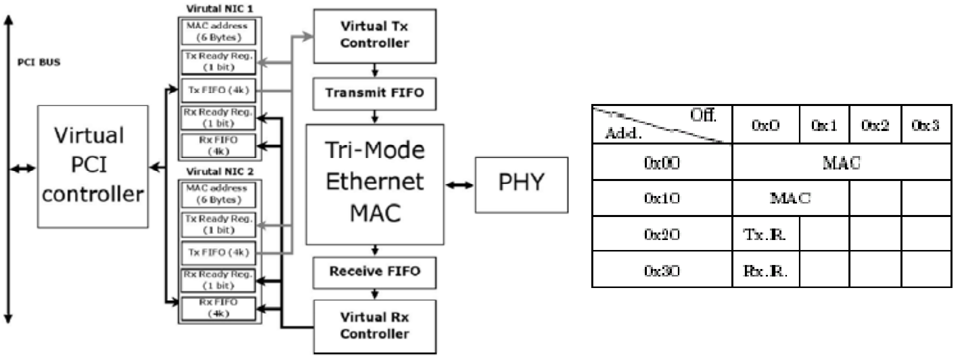
Figure 25: (a)x86 Memory Mapping, (b) PCI Device Driver Overview

The device driver functions in a Linux system as follows. The Linux kernel exports a general API that all the device drivers use. First, using the API, the device driver registers with the kernel. Registration enables the kernel to route user requests and interrupts to the device driver. After registration, the device driver searches for the PCI device and initializes it. This also is done using the kernel API.

3.2.8.2 Device Driver for the Virtual NIC

When an application uses the send() function to transmit a packet, the kernel invokes the transmit function associated with the NIC device driver. On the other hand, when a packet is received, the device interrupts the kernel, which in turn invokes the appropriate call back function to handle the interrupt. Interrupts are not used in this project. Instead, the device driver periodically checks the device to see if it has any pending incoming requests. To transfer data into the device, the device driver does a memcpy() from the kernel's memory to the device driver's memory. Programmed I/O was used to transfer data to and from the virtual devices.

The virtual NIC architecture depicted in Figure 26 (a) shows the registers in the NIC to which the driver has access. The first six bytes are reserved for the MAC address and they are read only. TX and RX registers are used to indicate the status of the NIC. If the Tx.R. register is a "1b," the NIC is busy; if it is a "0b," then the NIC is available. On the other hand, the Rx.R. register has a "1" if there is a valid packet in the data buffer and a "0b" otherwise.



Off. Add.	0x0	0x1	0x2	0x3
0x00	MAC			
0x10	MAC			
0x20	Tx.R.			
0x30	Rx.R.			

Figure 26: (a) Virtualized NIC’s Architecture, (b) Control Registers

Figure 27 (a) and (b) illustrates the transmit and receive state machines from the driver point of view. If the NIC driver gets a request to transmit a packet from the kernel, it checks the Tx.R. register to see if the NIC is ready, then it transfers the packet to the NIC and sets Tx.R. to “1b”. The Tx.R. will be cleared by the NIC when it transmits the packet. To receive a packet the driver will periodically poll the Rx.R register. If it has a “1b”, it will transfer the data to the kernel and set the Rx.R. to “0b”.

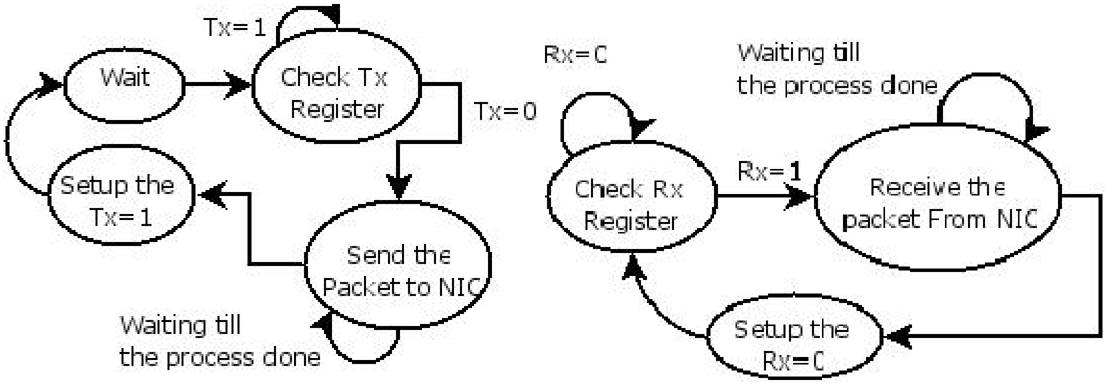


Figure 27: (a) VNIC Drivers TX State Machine, (b) VNIC Drivers RX State Machine

3.3 Secure Memory Manager (SMM)

Traditionally, x86 processors do not encrypt data and instructions stored in main memory. This raises security concerns when virtualizing these processors. Most Virtual Machine Monitor (VMM) vendors allow a Virtual Machine (VM) to be suspended and restored later. When a VM is suspended, its state information is stored as a file in the VM0's file system. An adversary could access this file to either steal or modify sensitive information. There have been attempts to encrypting memory contents either directly or by using a smart compiler/OS. In either case, the OS protects the keys by storing them in regions of the memory that cannot be accessed by the user. However, in a virtualized environment, when a VM is suspended, an adversary could read the keys from the memory dump file as the OS is no longer active.

An SMM architecture was designed so that, it protects the memory state of individual VMs by migrating this functionality into hardware (FPGA in our case). Encryption and decryption of the memory state of the VMs using per-domain-secret keys were implemented on the same FPGA. The memory used by the VMs is attached to the FPGA and is thus physically separate from the memory used by VM0 which is directly connected to the processor. Finally, appropriate hooks were developed in the device driver and Xen virtualization software.

3.3.1 Overview of Xen Memory Management

Modern operating systems use virtual memory to give an illusion to an application that it has a large amount of contiguous memory. The size is usually limited to the size of memory addressable directly by the CPU. For example, an application running in a 32-bit x86 CPU could virtually have up to 4GiB of memory. The virtual memory space is divided into pages and applications get a set of pages when they are executed. The page size is usually 4KiB in most operating systems. The operating system also sets up translation tables, known as page translation tables, for the CPU to translate virtual address to real physical address. When the application is running the CPU translates virtual address into physical address with the help of a Memory Management Unit (MMU). The MMU walks through the page translation tables to resolve a virtual address. Memory management in virtualized environments such as Xen requires one more translation on top of regular virtual to physical translation. Xen allocates pseudo-physical-memory to VM's when they are created. Xen maintains a pseudo-physical memory to physical memory mapping.   The pseudo-physical address space for each VM starts at “0x0”. The VM uses the pseudo-physical address range and is never exposed to the machine pages. All VMs have read-only access to the page tables. The VMM is notified of the reads and writes via hypercalls and validates them before upating the page tables. There are two page table modes under VMM.

Writable mode: As far as a VM is concerned it appears that the page tables are directly writable. VMM permits writes to pages and simultaneously invalidates the entry from an active page table. The dirty pages are re-connected either on context switches or when they are accessed again by the VM.

Shadow mode: The page tables in a VM have virtual-to-pseudo physical address mappings. These tables are never used by the hardware MMU. VMM maintains the Shadow Page Tables (SPT) in software. An SPT maps the virtual to machine addresses and MMU uses the SPT to detect any VM access to page tables.

SPT is dynamically generated from the VM page table and pseudo-physical-to-machine table. Hardware caches contain the recently accessed Virtual Page number (VPN)->Physical Machine Page Number (MPN) mapping. The VMM maintains a per-process VPN -> MPN table; the SPT and MMU make use of this table to resolve machine addresses. When a VM running on the processor performs a write to Page Table Base Register, it is trapped and scheduled out. VMM will make this base register point to the active process's SPT. Any updates to the VM page tables by that process also are trapped by the VMM and propagated to the process's SPT. The virtual addresses generated by the CPU while executing the process will be resolved via the process's SPT. In the shadow mode both the VM page table and SPT have to be up-to-date.

3.3.2 HAVEN SMM Architecture

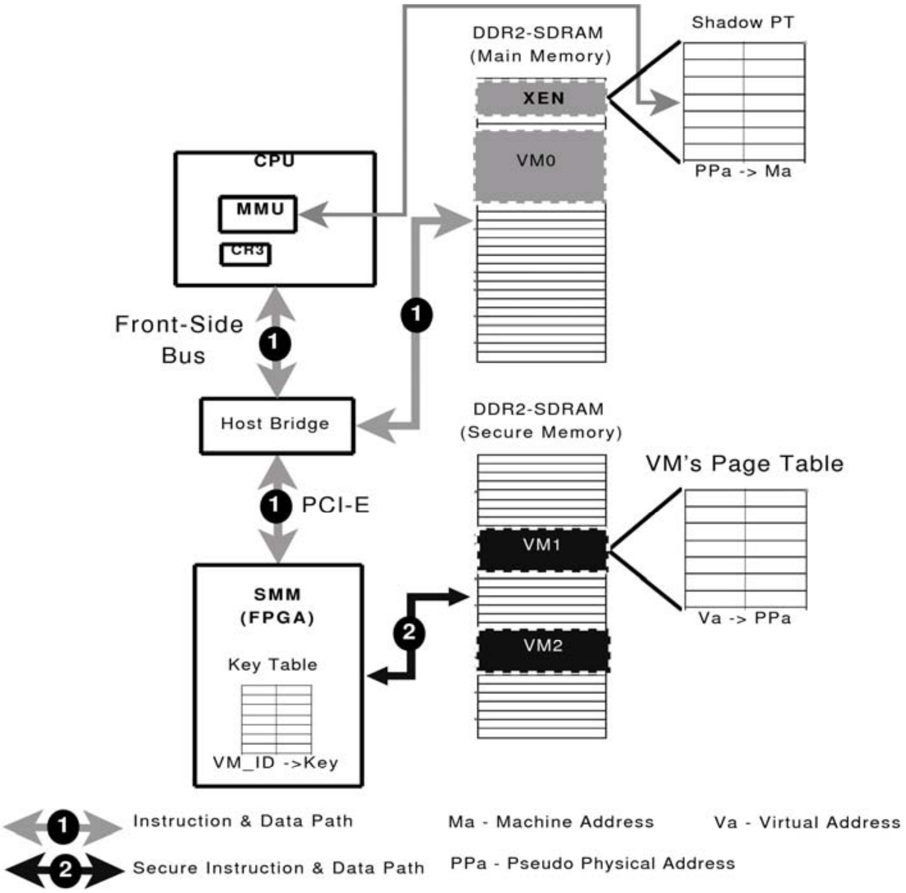


Figure 28: SMM High Level View

The SMM is implemented inside an FPGA co-processor which appears as a PCI device to Xen. Figure 28 depicts SMM hardware architecture. The memory controlled by the FPGA device is always encrypted and is allocated only to VMs. VM0 continues to operate from unencrypted memory that is directly accessible by the CPU. It also depicts how the SMM is connected to CPU. The SMM will track the active VMs and will know what VM is currently scheduled on a processor through VMM hooks. All memory requests by a VM are routed through the SMM on the FPGA which then decrypts the corresponding memory block. The secret keys supplied by an encryption decryption table are maintained by the SMM. The VMM operates in the shadow mode through which it detects writes to page tables by VMs. Any writes to page tables by VMs are trapped by the VMM which validates the writes to ensure secure isolation of VMs and notifies the VM whether the write is permitted or denied. Finally, the writes are propagated to the SPT.

In the HAVEN SMM architecture, the guest VMs are unaware of the encryption/decryption process. An administrative user can specify the encryption algorithm, the secret key length and other parameters. The SMM will only decrypt memory that is assigned to a VM. If the VMM or the VM0 tries to access memory that belongs to a guest VM, SMM will not decrypt the contents. However, SMM will not prevent access to the secure memory. When VM0 suspends a VM, it will be paused by the VMM and its encrypted memory contents (still in encrypted form) are dumped to a file. By default this feature will be enabled for all VMs.

3.3.3 SMM Implementation

Figure 29 shows the SMM implementation on a Xilinx ML507 FPGA board. We used the PCIe core and SDRAM controller provided by Xilinx. The SMM that we implemented includes starting from the left of the figure, the RX/TX controller, the SDRAM read/write controller, the secret key table and AES in counter mode. The key table is a lookup table that is populated offline. These keys are then used by AES to encrypt counters, which are subsequently XORed with data creating a stream cipher. Memory addresses are used to select the appropriate encryption and decryption keys.

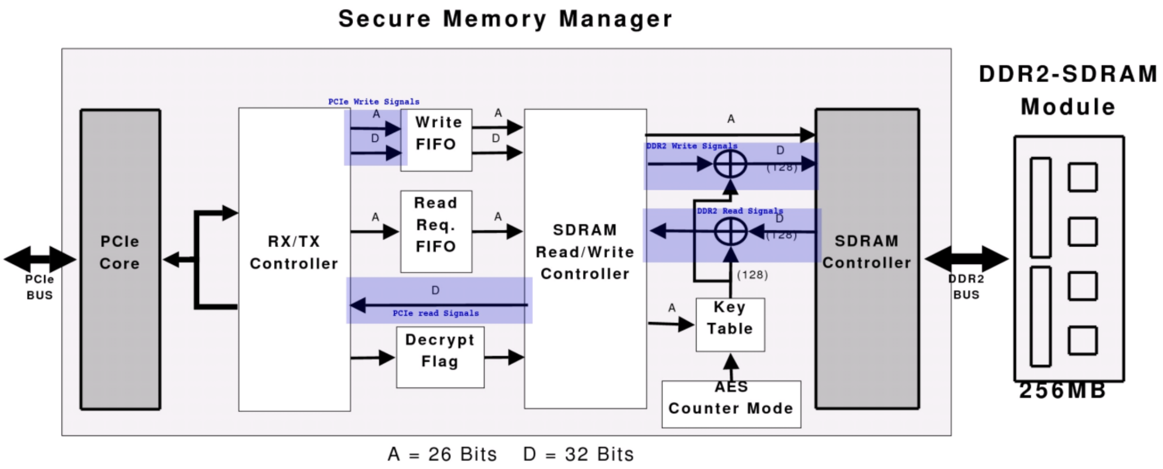
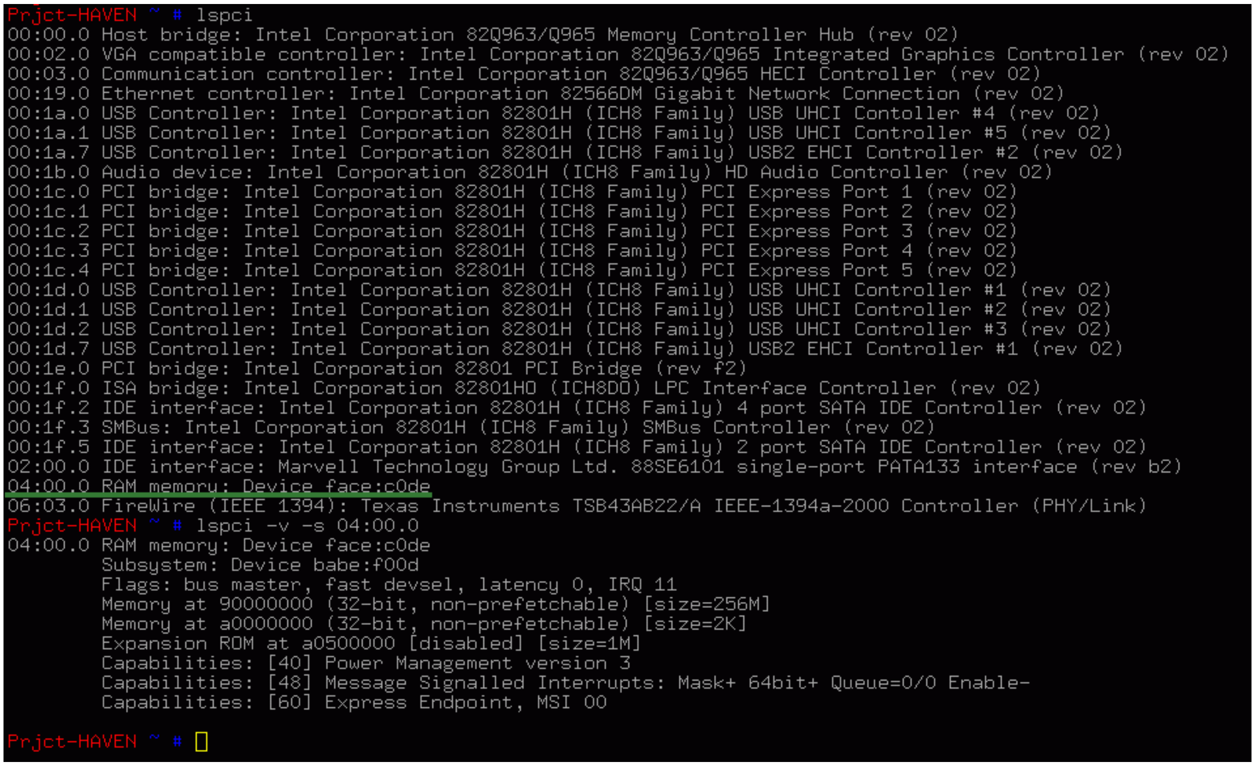


Figure 29: SMM Implementation

Our implementation of SMM and the SDRAM controller can operate on a 266 MHz. However, we are limited by the speed of the PCIe controller, which is 150 MHz.



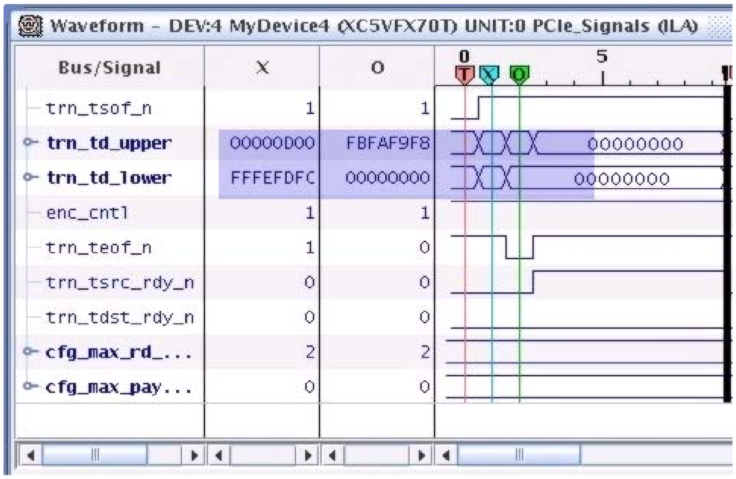
```
Prjct-HAVEN ~ # lspci
00:00.0 Host bridge: Intel Corporation 82Q963/Q965 Memory Controller Hub (rev 02)
00:02.0 VGA compatible controller: Intel Corporation 82Q963/Q965 Integrated Graphics Controller (rev 02)
00:03.0 Communication controller: Intel Corporation 82Q963/Q965 HECI Controller (rev 02)
00:19.0 Ethernet controller: Intel Corporation 82566DM Gigabit Network Connection (rev 02)
00:1a.0 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Contoller #4 (rev 02)
00:1a.1 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #5 (rev 02)
00:1a.7 USB Controller: Intel Corporation 82801H (ICH8 Family) USB2 EHCI Controller #2 (rev 02)
00:1b.0 Audio device: Intel Corporation 82801H (ICH8 Family) HD Audio Controller (rev 02)
00:1c.0 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 1 (rev 02)
00:1c.1 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 2 (rev 02)
00:1c.2 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 3 (rev 02)
00:1c.3 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 4 (rev 02)
00:1c.4 PCI bridge: Intel Corporation 82801H (ICH8 Family) PCI Express Port 5 (rev 02)
00:1d.0 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #1 (rev 02)
00:1d.1 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #2 (rev 02)
00:1d.2 USB Controller: Intel Corporation 82801H (ICH8 Family) USB UHCI Controller #3 (rev 02)
00:1d.7 USB Controller: Intel Corporation 82801H (ICH8 Family) USB2 EHCI Controller #1 (rev 02)
00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev f2)
00:1f.0 ISA bridge: Intel Corporation 82801HO (ICH8DO) LPC Interface Controller (rev 02)
00:1f.2 IDE interface: Intel Corporation 82801H (ICH8 Family) 4 port SATA IDE Controller (rev 02)
00:1f.3 SMBus: Intel Corporation 82801H (ICH8 Family) SMBus Controller (rev 02)
00:1f.5 IDE interface: Intel Corporation 82801H (ICH8 Family) 2 port SATA IDE Controller (rev 02)
02:00.0 IDE interface: Marvell Technology Group Ltd. 88SE6101 single-port PATA133 interface (rev b2)
04:00.0 RAM memory: Device face:c0de
06:03.0 FireWire (IEEE 1394): Texas Instruments TSB43AB22/A IEEE-1394a-2000 Controller (PHY/Link)
Prjct-HAVEN ~ # lspci -v -s 04:00.0
04:00.0 RAM memory: Device face:c0de
        Subsystem: Device babe:f00d
        Flags: bus master, fast devsel, latency 0, IRQ 11
        Memory at 90000000 (32-bit, non-prefetchable) [size=256M]
        Memory at a0000000 (32-bit, non-prefetchable) [size=2K]
        Expansion ROM at a0500000 [disabled] [size=1M]
        Capabilities: [40] Power Management version 3
        Capabilities: [48] Message Signalled Interrupts: Mask+ 64bit+ Queue=0/0 Enable-
        Capabilities: [60] Express Endpoint, MSI 00

Prjct-HAVEN ~ # 
```

Figure 30: Memory Detection

In the Figure 30 we show the presence of our Secure Memory Manager in a PC. Lspci is a utility that lists all PCI and PCIe devices that present in a system. Lspci with -v and -s <devcie ID> will list the details about a device. We can see that the we have created a RAM device and that device has 256MB of memory behind it.

3.3.3.1 PCIe Read

In the PCIe bus, a memory read is split into a memory read request and a read reply. The CPU issues a memory read request and the PCIe device replies to that request. Each request has a request ID and the CPU can issue more than one request before getting responses to a previous request. In the design the read request is stored in the Read Req. FIFO and when the requested data is available, a reply with requested data is generated by the RX/TX controller. SDRAM read/write controller monitors the Read Req. FIFO for pending requests and if SDRAM controller is not busy, it forwards the request to the SDRAM controller. SDRAM controller signals SDRAM read/write controller when the requested data is ready. Data is decrypted before passing it to the SDRAM read/write controller.

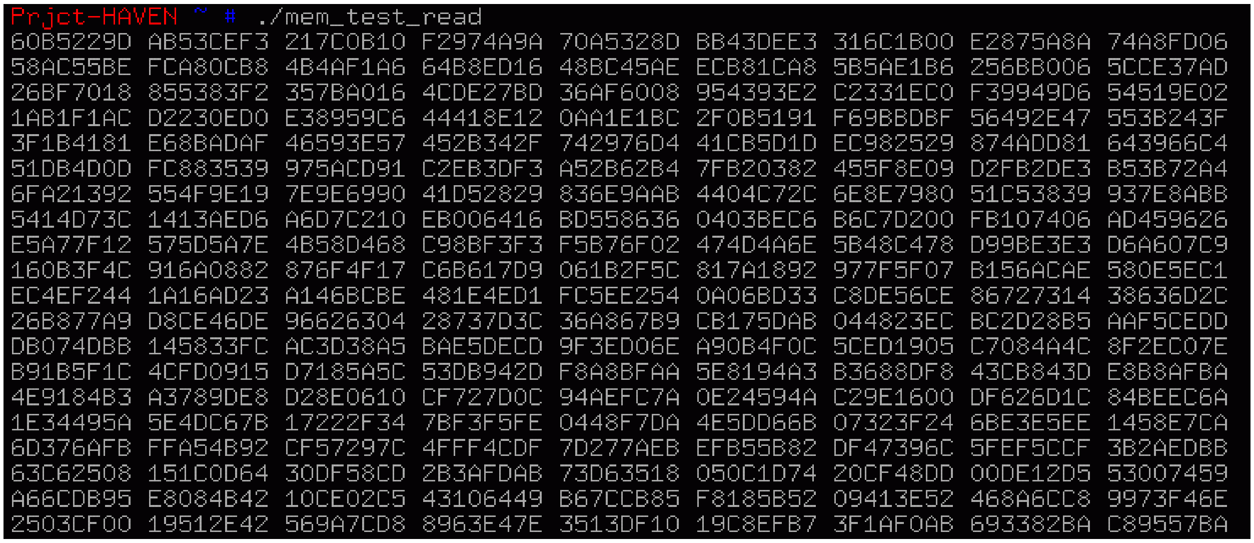


Waveform - DEV:4 MyDevice4 (XC5VFX70T) UNIT:0 PCIe_Signals (ILA)

Bus/Signal	X	O
trn_tsof_n	1	1
trn_td_upper	00000D00	FBFAF9F8
trn_td_lower	FFFEFDFC	00000000
enc_cntl	1	1
trn_teof_n	1	0
trn_tsrc_rdy_n	0	0
trn_tdst_rdy_n	0	0
cfg_max_rd_...	2	2
cfg_max_pay...	0	0

Figure 31: PCIe Read Cycle

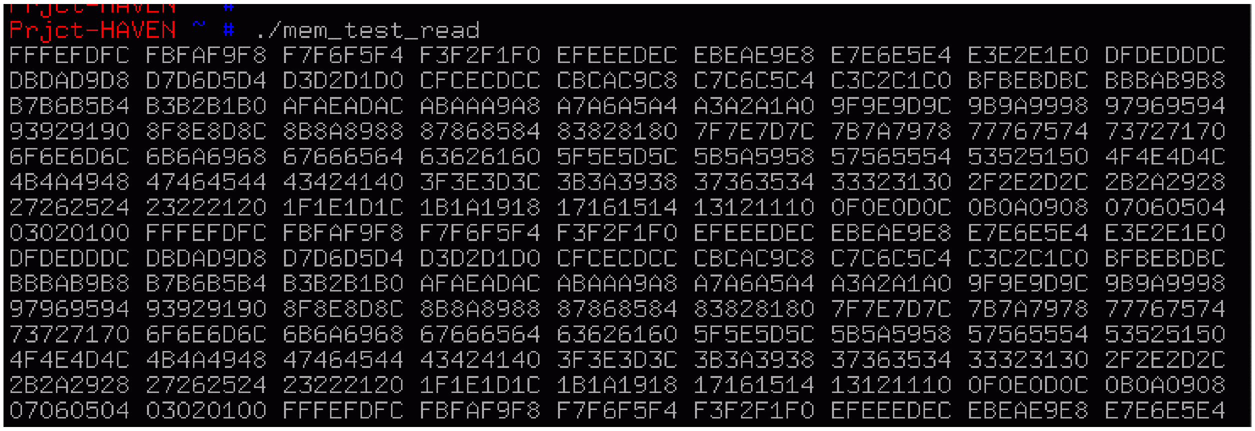
The shaded region in Figure 31 shows data read request reply signals. Two 32 bit data segments are being sent as reply for a read request. trn_td_upper carries the first 32-bit word and trn_td_lower carries the second 32-bit word. The core transfers 32-bits at a time. The snapshot in Figure 31 shows two such transfers (see “X” and “O”).



```
Prjct-HAVEN ~ # ./mem_test_read
60B5229D AB53CEF3 217C0B10 F2974A9A 70A5328D BB43DEE3 316C1B00 E2875A8A 74A8FD06
58AC55BE FCA80CB8 4B4AF1A6 64B8ED16 48BC45AE ECB81CA8 5B5AE1B6 256BB006 5CCE37AD
26BF7018 855383F2 357BA016 4CDE27BD 36AF6008 954393E2 C2331EC0 F39949D6 54519E02
1AB1F1AC D2230ED0 E38959C6 44418E12 0AA1E1BC 2F0B5191 F69BBDBF 56492E47 553B243F
3F1B4181 E68BADAF 46593E57 452B342F 742976D4 41CB5D1D EC982529 874ADD81 643966C4
51DB4D0D FC883539 975ACD91 C2EB3DF3 A52B62B4 7FB20382 455F8E09 D2FB2DE3 B53B72A4
6FA21392 554F9E19 7E9E6990 41D52829 836E9AAB 4404C72C 6E8E7980 51C53839 937E8ABB
5414D73C 1413AED6 A6D7C210 EB006416 BD558636 0403BEC6 B6C7D200 FB107406 AD459626
E5A77F12 575D5A7E 4B58D468 C98BF3F3 F5B76F02 474D4A6E 5B48C478 D99BE3E3 D6A607C9
160B3F4C 916A0882 876F4F17 C6B617D9 061B2F5C 817A1892 977F5F07 B156ACAE 580E5EC1
EC4EF244 1A16AD23 A146BCBE 481E4ED1 FC5EE254 0A06BD33 C8DE56CE 86727314 38636D2C
26B877A9 D8CE46DE 96626304 28737D3C 36A867B9 CB175DAB 044823EC BC2D28B5 AAF5CEDD
DB074DBB 145833FC AC3D38A5 BAE5DECD 9F3ED06E A90B4F0C 5CED1905 C7084A4C 8F2EC07E
B91B5F1C 4CFD0915 D7185A5C 53DB942D F8A8BFAA 5E8194A3 B3688DF8 43CB843D E8B8AFBA
4E9184B3 A3789DE8 D28E0610 CF727D0C 94AEFC7A 0E24594A C29E1600 DF626D1C 84BEEC6A
1E34495A 5E4DC67B 17222F34 7BF3F5FE 0448F7DA 4E5DD66B 07323F24 6BE3E5EE 1458E7CA
6D376AFB FFA54B92 CF57297C 4FFF4CDF 7D277AEB EFB55B82 DF47396C 5FEF5CCF 3B2AEDBB
63C62508 151C0D64 30DF58CD 2B3AFDAB 73D63518 050C1D74 20CF48DD 00DE12D5 53007459
A66CDB95 E8084B42 10CE02C5 43106449 B67CCB85 F8185B52 09413E52 468A6CC8 9973F46E
2503CF00 19512E42 569A7CD8 8963E47E 3513DF10 19C8EFB7 3F1AF0AB 693382BA C89557BA
```

Figure 32: Encrypted Memory Dump

In Figure 32 we show the encrypted memory dump of the data. This is what the SMM give out when an unauthorized VM tries to read the memory of another VM. As shown in the figure the the memory dump returns encrypted text.



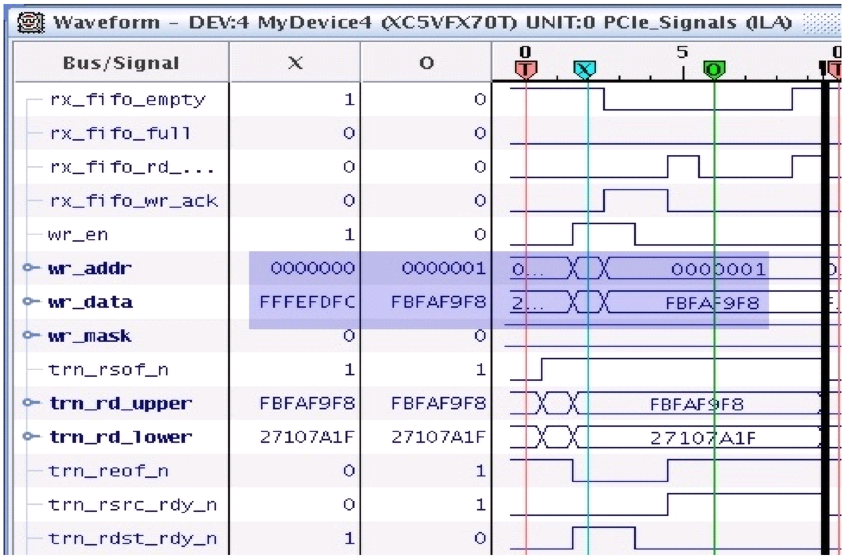
```
Prjct-HAVEN ~ # ./mem_test_read
FFFEFDFC FBFAF9F8 F7F6F5F4 F3F2F1F0 EFEEEDEC EBEAE9E8 E7E6E5E4 E3E2E1E0 DFDEDDDC
DBDAD9D8 D7D6D5D4 D3D2D1D0 CFCECDCC CBCAC9C8 C7C6C5C4 C3C2C1C0 BFBEBDBC BBBAB9B8
B7B6B5B4 B3B2B1B0 AFAEADAC ABAAA9A8 A7A6A5A4 A3A2A1A0 9F9E9D9C 9B9A9998 97969594
93929190 8F8E8D8C 8B8A8988 87868584 83828180 7F7E7D7C 7B7A7978 77767574 73727170
6F6E6D6C 6B6A6968 67666564 63626160 5F5E5D5C 5B5A5958 57565554 53525150 4F4E4D4C
4B4A4948 47464544 43424140 3F3E3D3C 3B3A3938 37363534 33323130 2F2E2D2C 2B2A2928
27262524 23222120 1F1E1D1C 1B1A1918 17161514 13121110 0F0E0D0C 0B0A0908 07060504
03020100 FFFEFDFC FBFAF9F8 F7F6F5F4 F3F2F1F0 EFEEEDEC EBEAE9E8 E7E6E5E4 E3E2E1E0
DFDEDDDC DBDAD9D8 D7D6D5D4 D3D2D1D0 CFCECDCC CBCAC9C8 C7C6C5C4 C3C2C1C0 BFBEBDBC
BBBAB9B8 B7B6B5B4 B3B2B1B0 AFAEADAC ABAAA9A8 A7A6A5A4 A3A2A1A0 9F9E9D9C 9B9A9998
97969594 93929190 8F8E8D8C 8B8A8988 87868584 83828180 7F7E7D7C 7B7A7978 77767574
73727170 6F6E6D6C 6B6A6968 67666564 63626160 5F5E5D5C 5B5A5958 57565554 53525150
4F4E4D4C 4B4A4948 47464544 43424140 3F3E3D3C 3B3A3938 37363534 33323130 2F2E2D2C
2B2A2928 27262524 23222120 1F1E1D1C 1B1A1918 17161514 13121110 0F0E0D0C 0B0A0908
07060504 03020100 FFFEFDFC FBFAF9F8 F7F6F5F4 F3F2F1F0 EFEEEDEC EBEAE9E8 E7E6E5E4
```

Figure 33: Decrypted Memory Dump

Figure 33 shows what the SMM give out when an authorized VM tries to read the memory. This time memory dump returns the decrypted text. As shown in the figure, the decrypted text contains numbers starting from “0xFF” to “0x00.”

3.3.3.2 PCIe Write

The memory write request does not require a reply. When a write request is received it is stored in the Write FIFO. The SDRAM read/write controller monitors the Write FIFO and forward write requests to the SDRAM controller when a request is available. Data is encrypted before passing it to the SDRAM controller.



Waveform - DEV:4 MyDevice4 (XC5VFX70T) UNIT:0 PCIe_Signals (ILA)

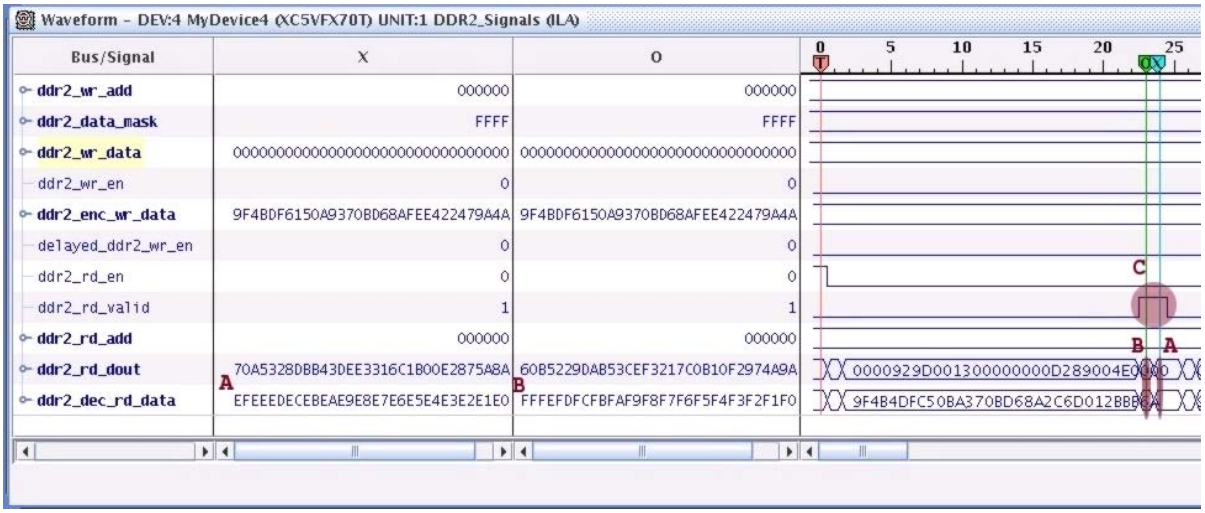
Bus/Signal	X	O
rx_fifo_empty	1	0
rx_fifo_full	0	0
rx_fifo_rd_...	0	0
rx_fifo_wr_ack	0	0
wr_en	1	0
wr_addr	0000000	0000001
wr_data	FFFEFDFC	FBFAF9F8
wr_mask	0	0
trn_rsof_n	1	1
trn_rd_upper	FBFAF9F8	FBFAF9F8
trn_rd_lower	27107A1F	27107A1F
trn_reof_n	0	1
trn_rsrc_rdy_n	0	1
trn_rdst_rdy_n	1	0

Figure 34: PCIe Write Cycle

The shaded blue region in
Figure 34 shows data (wr_data) “0xFFFEFDFC” and “0xFBFAF9F8” and address “0x0000000” and “0x0000001” (we_addr) coming from the PCIe core. The core transfers 32-bit at a time. The above capture shows two such transfers (see “X” and “O”).

3.3.3.3 DDR2 Read Operation

Figure 35 shows encrypted data coming out of SDRAM controller core. Here, ddr2_rd_dout shows the 128 bit data bus from SDRAM controller and ddr2_dec_rd_data is the 128 bit decrypted data.



Waveform - DEV:4 MyDevice4 (XC5VFX70T) UNIT:1 DDR2_Signals (ILA)

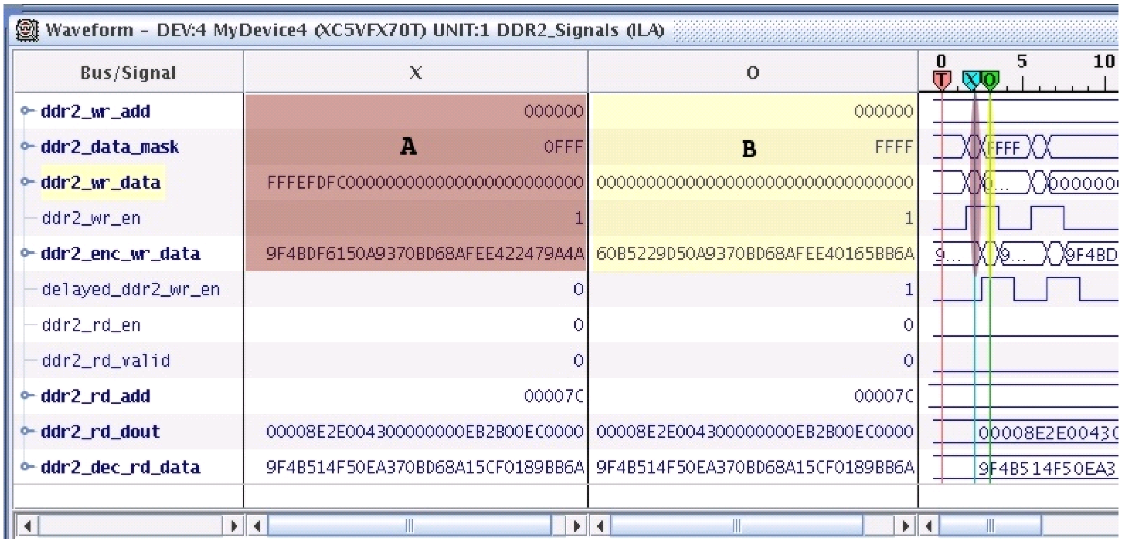
Bus/Signal	X	O
ddr2_wr_add	000000	000000
ddr2_data_mask	FFFF	FFFF
ddr2_wr_data	00000000000000000000000000000000	00000000000000000000000000000000
ddr2_wr_en	0	0
ddr2_enc_wr_data	9F4BDF6150A9370BD68AFEE422479A4A	9F4BDF6150A9370BD68AFEE422479A4A
delayed_ddr2_wr_en	0	0
ddr2_rd_en	0	0
ddr2_rd_valid	1	1
ddr2_rd_add	000000	000000
ddr2_rd_dout	70A5328DBB43DEE3316C1B00E2875A8A	60B5229DAB53CEF3217C0B10F2974A9A
ddr2_dec_rd_data	EFEEEDECEBEAE9E8E7E6E5E4E3E2E1E0	FFFEFDFCFBFAF9F8F7F6F5F4F3F2F1F0

0
5
10
15
20
25
T
O
X
C
B
A
A
B
0000929D00130000000D289004E0000
9F4B4DFC50BA370BD68A2C6D012BBB6A

Figure 35: DDR2 Read Operation

3.3.3.4 *DDR2 Write Operation*

Figure 36 shows clear text data “ddr2_wr_data” being encrypted “ddr2_enc_wr_data”. The SDRAM controller writes 256 bit blocks of data to SDRAM in every write cycle. Since PCIe could only transfer 32-bits at a time, we mask the 224-bit off to prevent data corruption.



Waveform - DEV:4 MyDevice4 (XC5VFX70T) UNIT:1 DDR2_Signals (ILA)

Bus/Signal	X	O
ddr2_wr_add	000000	000000
ddr2_data_mask	A 0FFF	B FFFF
ddr2_wr_data	FFFEFDFC000000000000000000000000	00000000000000000000000000000000
ddr2_wr_en	1	1
ddr2_enc_wr_data	9F4BDF6150A9370BD68AFEE422479A4A	60B5229D50A9370BD68AFEE40165BB6A
delayed_ddr2_wr_en	0	1
ddr2_rd_en	0	0
ddr2_rd_valid	0	0
ddr2_rd_add	00007C	00007C
ddr2_rd_dout	00008E2E004300000000EB2B00EC0000	00008E2E004300000000EB2B00EC0000
ddr2_dec_rd_data	9F4B514F50EA370BD68A15CF0189BB6A	9F4B514F50EA370BD68A15CF0189BB6A

Figure 36: DDR2 Write Operation

3.3.3.5 *Xen Hooks*

Proper functionality of SMM depends on the software hooks integrated into Xen’s VMM. Hooks must be implemented in three locations.

- Creation: When a VM is created, memory allocated to it should lie within the memory range controlled by the SMM. This is achieved by modifying Xen’s memory allocation routine populate_physmap(). The modified routine also updates SMM’s page permission table to permit access to memory for the newly created VM. Figure 37 (a) depicts how a VM is created and where the HAVEN hook’s are implemented.
- Scheduling: VMs are frequently scheduled out by the VMM for various reasons. Whenever a VM is scheduled out or scheduled in, SMM should be informed. To achieve this, Xen’s context_switch() routine is modified such that it updates SMM with the VM’s ID that is currently running. Figure 37 (b) depicts how VMs are scheduled and where the HAVEN hooks are implemented.

- Pause/Resume and Destruction: Similar to VM creation, memory must be de-allocated when a VM is paused or destroyed. HAVEN hooks are implemented in the decrease_reservation() routine in Xen's VMM to de-allocate memory behind SMM. The modifications simply remove entries from the page permission table in SMM. Figure 37(c) depicts how main memory and I/O devices memory are mapped.

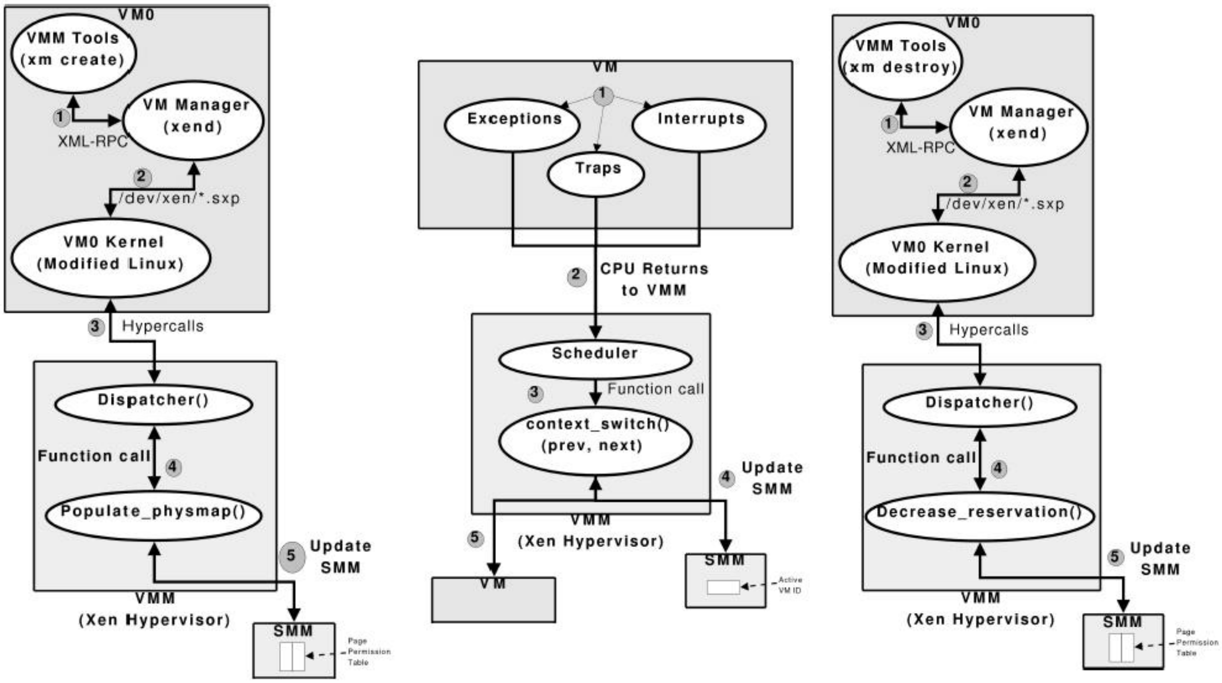


Figure 37: (a) Creation, (b) Context Switching (c) Destruction of a VM

4 CONCLUSIONS

We prototyped a High Assurance Virtualization ENgine (HAVEN) using Field Programmable Gate Array (FPGA) based secure co-processing to address the limitations of current virtualization technologies. HAVEN increases reliability via a hardware-assisted virtual I/O subsystem for each VM. It also improves performance by minimizing the switching back to the controller VM0 and by using a hardware virtual I/O manager. Furthermore, it improves security by protecting storage and communication channels using FPGA-assisted encryption and authentication.

We designed the HAVEN virtualization architectures for I/O and memory and implemented a proof of concept prototype. We developed VHDL and Verilog-HDL models, performed functional and timing simulations and validated the models using testbenches for the designed hardware primitives. We demonstrated the various functions of HAVEN virtualization prototype by developing Linux device drivers and Xen hooks.

5 REFERENCES

[1] Goldberg, Gerald J. Popek and Robert P. *Formal Requirements for Virtualizable Third Generation Architectures*. New York : ACM, 1974.

[2] Robin, John S. and Irvine, Cynthia E. *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*. Denver : USENIX Association, 2000, Vol. 9.

[3] The Honeynet Project. *http://www.honeynet.org*. [Online] 1999.

[4] Goldberg, Robert P. *Architectural Principles for Virtual Computer Systems*. 1973.

[5] *AMD64 Architecture. Programmer's Manual. Volume 2:. System Programming.* AMD64 Technology. 2007 .

[6] AMD. Codenamed. "Pacifica" Technology. Secure Virtual Machine. Architecture. 2005.

[7] Intel. *Intel® Virtualization Technology Specification for the IA-32 Intel® Architecture*. April 2005.

[8] Intel. *Intel® Virtualization Technology Specification for the Intel® Itanium® Architecture (VT-i)*. April 2005.

[9] Karger, Paul A. *Performance and security lessons learned from virtualizing the alpha processor*. s.l. : ACM, 2007.

[10] Ormandy, Tavis. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. *http://taviso.decsystem.org/virtsec.pdf*. [Online] 2007.

[11] Tal Garfinkel, Keith Adams, Andrew Warfield, Jason Franklin. *Compatibility is Not Transparency: VMM Detection Myths and Realities*. Berkeley : USENIX Association., 2007.

[12] Xen Multiple Vulnerabilities. *http://secunia.com/advisories/26986/*. [Online] 2007.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauery, Ian Pratt, Andrew Warfield. *Xen and the Art of Virtualization*. s.l. : ACM Press, 2003.

[14] XenSource. Xen. *http://xen.xensource.com/*. [Online] 2005.

[15] Sailer, Reiner, et al. *Secure Hypervisor Approach to Trusted Virtualized Systems*. s.l. : IBM, 2005. RC23511.

[16] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 179–188, New York, NY, USA, 2007.

6 SYMBOLS, ABBREVIATIONS AND ACRONYMS USED

VM	Virtual Machine
I/O	Input/Output
CPU	Central Processing Unit
HAVEN	High Assurance x86 Virtualization Engine
FPGA	Field Programmable Gate Array
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
VMM	Virtual Machine Monitor
NIC	Network Interface Card
MAC	Media Access Control
RX	Recieve
TX	Transmit
TEMAC	Tri-mode Ethernet Media Access Controller
SIM	Secure Virtual I/O manager
SMM	Secure Memory Manager
FIFO	First-in, First-out
OS	Operating System
MMU	Memory Management Unit
SPT	Shadow Page Tables
VPN	Virtual Page number
MPN	Machine Page Number